Techniques for Computing with Low-Independence Randomness

by

John Taylor Rompel

A.B., Computer Science and Mathematics
University of California, Berkeley
(1987)
S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1990

Signature of Author_____
Department of Electrical Engineering and Computer Science
September 6, 1990

Certified by_____
Frank Thomson Leighton
Professor of Applied Mathematics
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Techniques for Computing with Low-Independence Randomness

by

John Taylor Rompel

Submitted to the Department of Electrical Engineering and Computer Science
on September 6, 1990, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

During the past two decades, randomness has emerged as a central tool in the development of computational procedures. For example, there are many well known problems for which randomness can be used to design an algorithm that works well with high probability on *all* inputs, whereas the best known deterministic algorithms fail miserably on worst-case inputs. Unfortunately, generating truly random bits can be very difficult and/or expensive. This thesis shows that for several large and important classes of computational problems, however, a weaker and less expensive form of randomness—namely, k-wise independent distributions—will suffice. As a consequence, we are able to devise improved algorithms and cryptographic protocols for a wide variety of problems. The most important applications of this work are listed below.

- We obtain the first efficient parallel approximation algorithm for set cover. This linear processor, NC algorithm obtains a performance guarantee within a $(1+\epsilon)$ factor of the best sequential algorithm, while achieving a near-optimal speedup. This algorithm has applications in many areas, including parallel learning theory.
- We provide a general technique for removing randomness from parallel algorithms that depend on up to polylogarithmic independence. The technique substantially generalizes the *benefit function* framework of Luby to include functions which are a sum of a polynomial number of arbitrary functions of $O(\log n)$ boolean variables each. Special cases of polylogarithmic variable functions and multivalued random variables are also considered. As applications of these techniques, we provide the best known deterministic NC approximation algorithms for set discrepancy, weighted discrepancy, lattice approximation, edge coloring, and other problems.
- We give the first construction of a secure signature scheme that is based solely on the existence of one-way functions (where by secure we mean secure against existential forgery under adaptive chosen message attack). This improves upon the best previously known construction which requires the existence of a one-way permutation. Our construction is, in fact, optimal: the existence of one-way functions is both a necessary and sufficient condition for the existence of secure signature schemes. As part of constructing a signature scheme, we show how to construct a family of one-way hash functions given any one-way function.

Thesis Supervisor: Frank Thomson Leighton
Title: Professor of Applied Mathematics

Contents

Acknowledgments

I would like to thank my advisor, Tom Leighton. His insights, both technical and in presentation, are too many to count let alone list. Moreover, he has been extremely supportive of me throughout my stay at MIT. I will never forget how he stood behind me during the most trying times.

I would also like to thank the other two members of my thesis committee, Silvio Micali and Ron Rivest. Silvio taught me much of what I know about cryptography, and his brilliance inspired me to work in the field. In addition, he gave many helpful suggestions on the presentation of the signatures result. I am forever indebted to Ron for taking time out from his vacation to read my thesis, coming through for me where few would.

My time at MIT would have been radically different if not for Bonnie Berger. Her impeccable taste in problems has lead to three joint papers, two of which are in this thesis. Besides being a coauthor, she has been a close and loyal friend. I will miss her, although keeping in touch will be easier now that I have decided to stay in the same time-zone.

Mihir Bellare is another person who helped keep me sane at MIT. His wry humor has always been as close as the office next door. His imprint on this thesis is large: besides coauthoring the material on which Chapter 3 is based, he was extremely helpful with the signatures result, and also simplified the proof of Lemma 2.2.6 through the use of integrals.

Special thanks are in order for Lenore Cowen. Her help in editing this thesis was invaluable. I expect her to drive herself down to New York to see me very soon and only wish I could have been more helpful in this enterprise.

Thanks to Jill Burger who is always a friend, always kind, always willing to listen, and never dull and boring.

I would like to thank the National Science Foundation for funding my stay at MIT with a graduate fellowship, thus allowing me to devote my full energy towards research.

I would especially like to thank Neil Larson for facilitating and encouraging my early interest in computers. For over half my life, he has served as a mentor to me, and has been second only to my family as a source of support and encouragement.

My family is largely responsible for who I am. They always taught me to aim high. I am deeply grateful for all their love and support.

Introduction

Background

Randomization

Much research has been done on using randomness as an aid to computation. Many computational tasks become simpler if one assumes a computer has access to a device which "flips coins." Randomness can be used to "smooth out" problem instances so that, instead of having an application perform pathologically when given certain inputs, all inputs will be handled well almost all the time. In algorithmic applications, randomness is used to ensure that all instances are processed equally fast. In cryptographic applications, randomness is used to ensure that all inputs are equally well protected.

This smoothing process comes at a cost. Randomized algorithms require the availability of unbiased and independent random bits. However, getting the many independent "coin tosses" required for most randomized algorithms is slow and expensive. This is particularly true in the area of parallel algorithms. For a randomized parallel algorithm, we must generate many random bits very quickly. Although one can imagine adding a coin-flipping device to a single processor machine, it is more costly to add one to each processor of a multiprocessor computer.

Another case where generating independent random bits is costly is when we need a random function. In many cryptographic applications one is tempted to take the function one is working with and compose it with a random function in order to hide any structure which might be used by an adversary. However, if we are working with functions from n bits to n bits, a random function requires $n \cdot 2^n$ random bits to describe! In fact, even if generating the random bits were feasible, the length required to describe the function would make it infeasible to use.

This thesis will explore methods for using low-independence distributions as a way to produce random looking strings and functions using few truly random bits. Although these distributions use fewer random bits, we will be able to *prove* that the strings and functions we obtain are random enough to guarantee good performance for our applications. We can think of these distributions as a form of pseudorandom generator. Notice that we achieve a stronger form of pseudorandomness than is commonly considered. In practice, randomized algorithms are often run with a pseudorandom source which empirically looks random, e.g. it is hoped that the output of a linear congruential generator is sufficiently random to achieve good results. Cryptographers prove that their generators are random enough, but only under unproven complexity assumptions, e.g. they might prove that if the output can be distinguished from truly random bits then factoring is easy. In contrast, for the applications we consider, low-independence generators provably suffice.

k-wise Independence

Let us assume that we have an algorithm which flips n coins in the course of its execution. Another way to look at this is that our algorithm randomly picks one of the 2^n possible combinations of heads and tails which can occur. Perhaps, however, the algorithm may still work if we pick from a smaller collection of combinations if the smaller collection is somehow representative, representative in the sense that if you restricts your attention to any small subset of the coins, they appear random. More formally we say a collection of random variables $X_1, \ldots, X_n$ is *pairwise independent* if for any i and j, and any values a and b, the joint probability that X_i takes on value a and X_j takes on value b is the product of the individual probabilities of the two events. For example, distribution of three coins taken uniformly from the collection {HHH,HTT,THT,TTH} is pairwise independent since any two coins taken together have all four combinations equally likely. However, this distribution is not fully independent, since only four of the eight possible combinations of three coins can occur.

More generally, we say that random variables $X_1, \ldots, X_n$ are k-wise independent if for any k indices $i_1, \ldots, i_k$ and any k values $v_1, \ldots, v_k$,

$$\Pr[X_{i_1} = v_1 \text{ and } X_{i_2} = v_2 \text{ and } \cdots \text{ and } X_{i_k} = v_k] = \Pr[X_{i_1} = v_1] \Pr[X_{i_2} = v_2] \cdots \Pr[X_{i_k} = v_k].$$

This is useful since, in fact, there are pairwise independent collections of size $O(n)$, versus the

2^n required for full independence. In particular, this means that we need only $\log n$ random bits to pick from a pairwise independent distribution, versus n random bits for a fully independent distribution.

Major Results

Removing Randomness from Parallel Algorithms

An interesting consequence of the small size of k-wise independent distributions is that they can be used to remove randomness entirely. Consider an algorithm which flips n coins, but only requires that they be pairwise independent. Furthermore, assume the algorithm outputs either a valid solution or "failure." Karp and Wigderson [43] noted that one could run the algorithm for each of the $O(n)$ different combinations in a pairwise independent distribution. Since the randomized algorithm works with non-zero probability, one of the $O(n)$ combinations will yield a good solution. In fact, we can run the different copies of the algorithm in parallel, thus increasing only the number of processors required—not the running time. This method can be extended to remove the randomness from any parallel algorithm depending only on constant independence [2, 48]. More recently, Luby [49] developed a method for removing the randomness from parallel algorithms depending only on pairwise independence which does not increase the number of processors used. To do this, he shows how to perform a binary search on a particular pairwise independent distribution.

In Chapter 1 of this thesis, we will develop the first parallel algorithms for approximate set cover. We will begin by developing a randomized parallel algorithm which requires only pairwise independence. Even this is surprising, since the known algorithm was of a greedy nature, where each choice depends on all the previous choices, and thus does not seem amenable to parallelization. We will then apply the techniques of Luby to convert it to a linear processor deterministic parallel algorithm. Our algorithms for set cover have applications to computational learning theory and computational geometry.

In Chapter 2, we will extend the binary search techniques of Luby to problems using $(\log n)$-wise independence. To solve these problems using exhaustive search can be shown to require a superpolynomial number of processors [2, 22]. Thus we are able to give the first efficient deter-

ministic parallel algorithms for several problems, most notably set discrepancy, edge coloring, hypergraph coloring, and lattice approximation.

Hash Functions

The other main application of low-independence is to *universal hash functions* [19]. Universal hash functions are families of functions where the individual function values are k-wise independent random variables (for some k). These look random in many respects. For example, with a pairwise universal hash function, the probability that two elements of the domain will collide is the same as if a truly random function was used. On the other hand, they have concise descriptions; a k-wise independent function from n-bits to m-bits requires only $k \cdot \max(n, m)$ bits to describe.

Most work using universal hash functions has concentrated on the pairwise independent variety. In this thesis, we will develop techniques for using higher independence families.

In Chapter 3, we will consider the problem of amplifying Arthur-Merlin games. These are protocols where an infinite power prover interacts with a polynomial time verifier. These typically have an error probability of 1/3. It is well-known that if $O(n)$ copies of the protocol are executed in parallel, the error probability is reduced to 2^{-n}. However, this increases the number of random bits needed by a factor of n.

We reduce the amplification problem to one of "obliviously sampling," namely getting a collection of x's such that the average value of a function f applied to them is close to the expected value of $f(x)$, without actually computing f. We can think of sampling as composing f with a random hash function, i.e. take $g(x) = f(h(x))$, then considering $g(1), g(2), \ldots, g(n)$. We show a sharp tradeoff between the quality of the sample, the size of the sample, and the independence. As a first result, we show that an $(n/\log n)$-wise independent hash function suffices, thereby giving an immediate $\log n$ reduction in the number of random bits required. As a more sophisticated method, we can think of taking a series of samples (implemented by composing a series of hash functions), each time shrinking the size but increasing the independence. By doing this we are able to achieve a good sample using far fewer random bits.

In Chapter 4, we will explore the topic of secure digital signatures. We show that a secure signature scheme can be constructed from any one-way function (a function which is easy to

compute, but hard to invert). Previously, it was only known how to construct a signature scheme given a one-way permutation. To build a signature scheme from an arbitrary one-way function, we construct a series of functions, beginning with our original one-way function, and each having a slightly stronger cryptographic property. Many of the steps consist of compositions with universal hash functions. It is most convenient to think of these as random functions used to smooth things out. However, we need to have a polynomial length description of whatever function we use. Thus it is important to show that low independence suffices. It is worth noting that while some of these steps are based on pairwise independence, most require the sharper bounds obtained by greater independence.

Chapter 1

Parallel Algorithms for Approximate Set Cover

1.1 Introduction

Given a hypergraph $H = (V, E)$ with $|V| = n$ and $|E| = m$, and a cost function on the vertices $c : V \to \mathbf{R}$, the *weighted set cover* problem consists of finding a minimum cost subset $R \subseteq V$ which covers H; i.e., an R that minimizes $c(R) = \sum_{v \in R} c(v)$ subject to the constraint that $e \cap R \neq \emptyset$ for all $e \in E$. This is equivalent to the problem of, given a set system $\mathcal{A} \subseteq 2^X$ and a cost function $c : \mathcal{A} \to \mathbf{R}$, finding a minimum cost subcollection $\mathcal{A}' \subseteq \mathcal{A}$ such that $\bigcup \mathcal{A}' = X$. The set cover problem is *NP*-complete [42], so we will not be concerned with algorithms giving exact solutions. Instead, we will consider *approximation algorithms*, algorithms that output sub-optimal solutions with a *performance guarantee* bounding the worst-case ratio between the cost of the solution output and the optimal solution.

The best known polynomial-time approximation algorithm for set cover is the greedy set cover algorithm [38, 47, 23]. Surprisingly, showing that the greedy algorithm performs well is fairly challenging, as is evident in the proofs of Johnson, Lovász, and Chvatal. They show that the greedy algorithm has a $(1 + \ln \Delta)$ performance guarantee (i.e. it always produces a cover of cost at most $(1 + \ln \Delta)$ times optimal), where Δ is the maximum degree of H (i.e.

This chapter describes joint work with Bonnie Berger and Peter Shor [15].

the maximum number of edges containing any node). However, the greedy algorithm seems inherently sequential. Although *RNC* algorithms have been proposed which perform well for some special cases [20, 3, 63], until now no parallel algorithm which performs well on arbitrary instances has been developed.

The main result of this chapter is a linear-processor deterministic *NC* algorithm that always finds a cover which is within a $(1+\epsilon)\log\Delta$ factor of the optimal cost. Hence, the algorithm achieves virtually the same performance as the best sequential algorithm in terms of cover size, and is within a polylogarithmic factor in terms of processor-time product. To obtain an algorithm that uses only a linear number of processors, we first give a randomized algorithm that only needs pairwise independence to guarantee a good solution and then adapt the techniques of Luby [49] to derandomize and get a linear-processor deterministic algorithm.

The set cover algorithms we develop have applications to parallel learning theory. We consider a well-known learning problem that has been solved in the sequential domain [18], and solve it in parallel. In particular, we consider the problem of learning in concept classes that are formed by taking either finite unions or finite intersections of a fixed base class of finite VC dimension. We show that classes of this type are *NC*-learnable whenever there is an *NC* algorithm for finding a consistent hypothesis in the base class. The only previous work on parallel learning is in [63]. They give an *RNC* algorithm for learning s-fold unions of axis-parallel rectangles in the plane, by using a randomized set cover algorithm for specialized hypergraphs. Our general techniques solve this problem as a special case. In addition, while [63, 64] may produce a hypothesis with up to $O(s^2 \log m)$ rectangles (where m is the sample size), our method will always produce one with at most $O(s \log m)$ rectangles, which is within a logarithmic factor of optimal.

The remainder of this chapter is divided into sections as follows. In Section 1.2, we present an *RNC* algorithm for approximate set cover. In Section 1.3, we review the known derandomization techniques for parallel algorithms and show how to use them to remove the randomness from our *RNC* algorithm, obtaining an *NC* algorithm for approximate set cover. In Section 1.4, we show how to use set cover, as well as other tools, to solve the learning problem described above.

1.2 Obtaining an *RNC* Algorithm for Approximate Set Cover

In this section we show how to obtain an *RNC* approximation algorithm for Set Cover which achieves a performance guarantee within a $1+\epsilon$ factor of the best known sequential algorithm. The analysis of our algorithm will depend only upon on pairwise independence. This will allow us, in Section 1.3, to convert our algorithm into a deterministic *NC* algorithm obtaining the same performance guarantee.

1.2.1 Emulating The Greedy Algorithm

The best known polynomial time approximation algorithm for set cover is the greedy algorithm, which was developed independently by Johnson and Lovász for the unweighted case [38, 47], and was extended to the weighted case by Chvatal [23]. The greedy algorithm is as follows: given hypergraph $H = (V, E)$, and cost function $c : V \rightarrow \mathbf{R}$, we pick the vertex with minimum cost per edge covered (i.e. minimum $c(v)/d(v)$, where $d(v)$ is the number of edges containing vertex v). We add it to the cover, remove it and all edges containing it from the hypergraph, and repeat until there are no edges left.

Define a fractional cover of H to be a function $f : V \rightarrow \mathbf{R}$ such that $0 \leq f(v) \leq 1$ for all $v \in V$ and $\sum_{v \in e} f(v) \geq 1$ for all $e \in E$. (Note that a normal cover is a fractional one where $f(v) \in \{0, 1\}$ for all $v \in V$.) The cost of a fractional cover is $\sum_{v \in V} f(v)c(v)$. Chvatal proved:

Theorem 1.2.1 ([23]) *The greedy algorithm outputs a cover R of H with $c(R) \leq (\ln \Delta + 1)\tau^*$, where τ^* is the cost of the optimal fractional cover of H.*

Unfortunately, the greedy algorithm seems inherently sequential in nature. The degree of each vertex depends on which vertices have already been picked for the cover. Thus, which vertex we pick at any step depends on which vertices were picked at all prior steps.

However, we can devise parallel algorithms which are close to the greedy algorithm. We will seek to, instead of picking one vertex at a time, pick a large collection of vertices which cover almost as many edges as the greedy algorithm could picking that many vertices one at a time.

Definition 1.2.2 *An algorithm* α-emulates *the greedy algorithm if it outputs a cover in a series of steps, each of which picks a set of vertices R with the property that the total cost of R divided*

by the number of edges covered by R is at most α times the minimum cost per edge at the start of the step, i.e.

$$\frac{c(R)}{|\{e : e \cap R \neq \emptyset\}|} \leq \alpha \min_{v \in V} \frac{c(v)}{d(v)}$$

That this notion of emulating the greedy algorithm is good enough to give good approximations is captured by the following theorem.

Theorem 1.2.3 *Any algorithm which α-emulates the greedy algorithm will output a cover R of H with $c(R) \leq \alpha(\ln \Delta + 1)\tau^*$, where τ^* is the cost of the optimal fractional cover of H.*

Proof Let f be an optimal fractional cover, i.e. a function $f : V \rightarrow [0,1]$ such that $\sum_{v \in V} f(v)c(v) = \tau^*$ and, for all $e \in E$, $\sum_{v \in e} f(v) \geq 1$. To prove the stated bound, we will show how to "simulate" f using the cover produced by our algorithm.

For each $e \in E$, we will let y_e be the cost of covering edge e. More specifically, if the selection step which covered e did so by selecting P and k edges were covered by P, then y_e would be set to $c(P)/k$. Clearly, the cost of the cover produced is

$$\begin{aligned} \sum_{e \in E} y_e &\leq \sum_{e \in E} \sum_{v \in e} f(v) y_e \\ &= \sum_{v \in V} \left(f(v) \sum_{e \in E_v} y_e \right), \end{aligned}$$

where $E_v = \{e \in E | v \in e\}$. We can think of the inner sum, $\sum_{e \in E_v} y_e$, as the cost of simulating v with our cover (i.e. it is the price we paid to cover the edges that v covers). In the remainder of this proof we will show that $\sum_{e \in E_v} y_e$ is bounded above by $\alpha(1 + \ln \Delta)c(v)$, from which the theorem follows.

We claim that for any v and b, the number of edges $e \in E_v$ with $y_e \geq b$ is at most $\alpha c(v)/b$. Assume this were not true for some v and b. Then at the point in the execution of the algorithm just before the first $e \in E_v$ with $y_e \geq b$ is covered, the current degree of v is more than $\alpha c(v)/b$. But this implies that at that point in the algorithm,

$$\begin{aligned} \frac{c(R)}{|\{e : e \cap R \neq \emptyset\}|} &= y_e \\ &\geq b \\ &> \alpha c(v)/d(v) \\ &\geq \alpha \min_{v \in V} \frac{c(v)}{d(v)}, \end{aligned}$$

contradicting the assumption that the algorithm α-emulates the greedy algorithm.

From the above claim, we get that

$$\begin{aligned}\sum_{e \in E_v} y_e &\leq \sum_{i=1}^{d(v)} \alpha c(v)/i \\ &= \alpha c(v) \sum_{i=1}^{d(v)} 1/i \\ &\leq \alpha c(v)(1 + \ln d(v)) \\ &\leq \alpha c(v)(1 + \ln \Delta),\end{aligned}$$

as promised. □

1.2.2 A Special Case

In the previous section, we showed that any algorithm that, for some small α, α-emulates the greedy algorithm, will achieve a good performance guarantee for set cover. To obtain an *(R)NC* algorithm, we must now show we can α-emulate the greedy algorithm in a polylogarithmic number of steps. This means many of these selection steps must pick a very large number of vertices simultaneously and yet perform almost as well as the greedy algorithm would, picking that many vertices one at a time. In this section, we will demonstrate a simple case where this is easily done. Then in the next section, we will show how to extend these techniques to work in general.

Here we will consider the special case of a hypergraph where all edges have the same number of vertices, all vertices are contained in the same number of edges, and the cost of every vertex is 1. Let $H = (V, E)$ be a hypergraph, $|V| = n$, and $|E| = m$. Furthermore assume that for each $e \in E$, $|e| = a$ and that each vertex in V is contained in exactly b edges of E. These conditions, in particular, will imply that $ma = nb$.

Let δ be a small positive constant between 0 and 1. We will consider randomly picking vertices with the goal of covering about a δ fraction of the edges. In particular, we will pick each vertex with probability $p = \delta/a$, independently of all other vertices. This immediately gives an expected number of vertices picked of pn and an expected number of edge coverings (where if an edge has two of its vertices picked, it is counted twice, etc.) of $map = \delta m$.

To count the expected number of edges covered, we look at the probability that a single edge is covered. Without loss of generality, assume we are considering edge $e = \{v_1, \ldots, v_a\}$, and let X_i be the event that vertex v_i is picked. Then,

$$\begin{aligned}
\Pr[e \text{ is covered}] &= \Pr[X_1 \vee X_2 \vee \cdots \vee X_a] \\
&= 1 - \Pr[\bar{X}_1 \wedge \bar{X}_2 \wedge \cdots \wedge \bar{X}_a] \\
&= 1 - (1-p)^a \\
&\geq 1 - e^{-\delta} \\
&\geq \delta - \delta^2/2.
\end{aligned}$$

Thus, by linearity of expected value, the expected number of edges covered is at least $(1 - \delta/2)\delta m$.

Now consider the greedy algorithm picking pn vertices, one at a time. Since every vertex is in exactly b edges, the best the greedy algorithm (or any algorithm, for that matter) could do is to cover pnb edges. Simplifying, we find $pnb = \delta m$. So our randomized algorithm covered almost a δ fraction of the edges, emulating the greedy algorithm within a $1/(1 - \delta/2)$ factor. If we could do this in general, we could cover all of the the edges using only $\log_{(1+\delta)} m$ selection steps, and thus would have a good *RNC* approximation algorithm for set cover. Unfortunately, in the above example, we needed the hypergraph to be in a very special form. In the next section we will show how to handle general hypergraphs in a series of selection steps resembling this one.

Before showing the general result, we will show an alternative analysis of the simple result which will require only pairwise independence. This form is actually more convenient to use in the next section; it will also allow us to derandomize and obtain *NC* algorithms in Section 1.3.

Recall that we were lower bounding $\Pr[X_1 \vee X_2 \vee \cdots \vee X_a]$. Rather than apply DeMorgan's Law, we can lower bound this by the first two terms of the inclusion-exclusion expansion; namely,

$$\begin{aligned}
\Pr[X_1 \vee X_2 \vee \cdots \vee X_a] &\geq \sum_{1 \leq i \leq a} \Pr[X_i] - \sum_{1 \leq i < j \leq a} \Pr[X_i \wedge X_j] \\
&= ap - \binom{a}{2} p^2 \\
&\geq \delta - \delta^2/2.
\end{aligned}$$

Notice, whereas before we needed full independence to say $\Pr[\bar{X}_1 \wedge \bar{X}_2 \wedge \cdots \wedge \bar{X}_a] = (1-p)^a$, here we only need pairwise independence to say that for all i and j, $\Pr[X_i \wedge X_j] = p^2$. The rest of the analysis is identical to before.

1.2.3 The *RNC* Algorithm

Now we return to the problem of finding a near optimal cover for an arbitrary hypergraph. We proceed as follows. Let ϵ be a constant, $0 < \epsilon \leq 1/12$. We cover H in a series of stages. At the beginning of stage h (stages are sequenced in decreasing order), the hypergraph induced by the algorithm thus far has maximum degree per unit cost less than $(1+\epsilon)^h$. During this stage, we restrict our attention to the subhypergraph induced by the vertices of degree per unit cost between $(1+\epsilon)^{h-1}$ and $(1+\epsilon)^h$ and only add these vertices to the cover. In this way we only add to the cover vertices that have degree per unit cost close to the maximum, thereby emulating the greedy algorithm.

It will be necessary in our proofs that all the vertices we handle have roughly the same degree. To ensure this, we divide each stage into phases. During phase i (again, phases are sequenced in decreasing order), we restrict the hypergraph to vertices of degree between $(1+\epsilon)^{i-1}$ and $(1+\epsilon)^i$. At the beginning of phase i, all vertices have degree at most $(1+\epsilon)^i$. At the end of this phase, all unpicked vertices will have either degree at most $(1+\epsilon)^{i-1}$ or degree per unit cost at most $(1+\epsilon)^{h-1}$.

In the previous section, since every edge was of size a, we could pick vertices with probability δ/a with the result that we covered a large fraction of the edges without covering many edges more than once. Here, however, the edge sizes can vary arbitrarily, making the task of getting a single probability impossible. If we set the probability too low, then the small edges will not be hit quickly enough. If we set it too high, then the large edges will be hit many times; this is undesirable since the average number of edges covered by each vertex could become much smaller than the maximum degree, thus deviating from the behavior of the greedy algorithm. Our solution is to perform a sequence of subphases. At the beginning of subphase j (again sequenced in decreasing order), all edges contain fewer than $(1+\epsilon)^j$ vertices (which were restricted above to have degree per unit cost between $(1+\epsilon)^{h-1}$ and $(1+\epsilon)^h$ and degree between $(1+\epsilon)^{i-1}$ and $(1+\epsilon)^i$). During this subphase, we repeatedly pick vertices with

probability $\delta/(1+\epsilon)^j$, where $0 < \delta \leq 1/12$. Picking vertices with this probability allows us to cover a $\delta/2$ fraction of the edges of size at least $(1+\epsilon)^{j-1}$, but does not cause many edges to be hit more than once, since no edge is larger than $(1+\epsilon)^j$.

More precisely, a subphase consists of a series of selection steps, performed until there are no more edges of size at least $(1+\epsilon)^{j-1}$. The selection steps are of two types. If some vertex covers a $\delta^3/(1+\epsilon)$ fraction of these large edges, we select such a vertex. Otherwise, we run a selection procedure (given below) that produces a collection P of vertices covering at least $c(P)(1+\epsilon)^h(1-6\delta-2\epsilon)$ edges, including at least a $\delta/2$ fraction of the large edges. In both cases, the selected vertex or vertices are added to the cover and deleted; the edges covered by these are deleted; and vertices that now have degree per unit cost less than $(1+\epsilon)^{h-1}$ (or degree less than $(1+\epsilon)^{i-1}$) are removed from consideration for this stage (or phase).

Clearly this algorithm $\frac{1}{1-6\delta-2\epsilon}$-emulates the greedy algorithm, and thus will produce a near-optimal cover. It remains to show how to perform the selection procedure, and that the algorithm is in *RNC*.

Now we give a randomized version of the selection procedure. Let $H_i = (V_i, E_i)$ be the current hypergraph, restricted to vertices of degree per unit cost between $(1+\epsilon)^{h-1}$ and $(1+\epsilon)^h$ and degree between $(1+\epsilon)^{i-1}$ and $(1+\epsilon)^i$. Let $E_{ij} \subseteq E_i$ contain the edges of E_i that have at least $(1+\epsilon)^{j-1}$ vertices. We are given that no vertex in V_i covers more than a $\delta^3/(1+\epsilon)$ fraction of E_{ij}. We want to return a $P \subseteq V_i$ such that P covers at least $c(P)(1+\epsilon)^h(1-6\delta-2\epsilon)$ edges, including at least a $\delta/2$ fraction of E_{ij}. The randomized algorithm generates P by including each vertex of V_i with probability $\delta/(1+\epsilon)^j$, pairwise independently. If P is good, we return it, otherwise we try again.

Lemma 1.2.4 *With probability at least* $1/8$*, a random* P *is good, i.e.* P *covers at least* $c(P)(1+\epsilon)^h(1-6\delta-2\epsilon)$ *edges, including at least a* $\delta/2$ *fraction of* E_{ij}*.*

Proof To show P covers close to $c(P)(1+\epsilon)^i$ edges, it suffices to show that P covers many edges and that $c(P)$ is not too large. Because of our restriction to H_i, we know that $c(P) \leq (1+\epsilon)^{i-h+1}|P|$, so it suffices to show that $|P|$ is not too large. We let X_i be 1 if v_i is picked, and 0 otherwise. In terms of our random variables, $|P| = \sum_{k \in V_i} X_k$, which we denote by $f_1(X)$. We will show that with probability 3/4, $f_1(X) \leq (\delta + 2\delta^2)|V_i|/(1+\epsilon)^j$; i.e., $f_1(X) - E[f_1(X)] \leq$

$2\delta^2|V_i|/(1+\epsilon)^j$. Let us note that the variance of $f_1(X)$ is at most $|V_i|\delta/(1+\epsilon)^j$. Moreover, let us observe that $|V_i| > (1+\epsilon)^j/\delta^3$, since no vertex in V_i covers a $\delta^3/(1+\epsilon)$ fraction of E_{ij}. Therefore, we know by Chebyshev's inequality that with probability at least 3/4,

$$(f_1(X) - E[f_1(X)])^2 \leq 4\mathrm{Var}(f_1(X)) \leq 4|V_i|\frac{\delta}{(1+\epsilon)^j} \leq 4|V_i|^2\frac{\delta^4}{(1+\epsilon)^{2j}},$$

which implies

$$|f_1(X) - E[f_1(X)]| \leq 2|V_i|\frac{\delta^2}{(1+\epsilon)^j}.$$

A lower bound on the number of edges covered by P is

$$\sum_{e\in E_i}(\sum_{k\in e} X_k - \sum_{k,l\in e} X_kX_l) = \sum_{k\in V_i} d(k)X_k - \sum_{e\in E_i}\sum_{k,l\in e} X_kX_l.$$

The first term is bounded below by $|P|(1+\epsilon)^{i-1}$, which is at least $(\delta - 2\delta^2)|V_i|(1+\epsilon)^{i-j-1}$, if $(f_1(X) - E[f_1(X)])^2 \leq 4\mathrm{Var}(f_1(X))$ as before. Denote the second term by $f_2(X)$. The expectation of f_2 is at most $((1+\epsilon)^i|V_i|/(1+\epsilon)^j)\,\frac{\delta^2}{2}$. Thus with probability at least 3/4,

$$f_2(X) \leq 4E[f_2(X)] \leq 2\delta^2(1+\epsilon)^{i-j}|V_i|.$$

With probability at least 1/2, both events hold, so

$$\begin{aligned} c(P) &\leq |P|(1+\epsilon)^{i-h+1} \\ &\leq (\delta + 2\delta^2)|V_i|(1+\epsilon)^{i-h-j+1} \end{aligned}$$

and the number of edges covered is at least

$$\begin{aligned} (1+\epsilon)^{i-j-1}|V_i|(\delta - 4\delta^2 - 2\delta^2\epsilon) &\geq c(P)(1+\epsilon)^{h-2}\frac{\delta - 4\delta^2 - 2\delta^2\epsilon}{\delta + 2\delta^2} \\ &\geq c(P)(1+\epsilon)^h(1 - 6\delta - 2\epsilon). \end{aligned}$$

Now we consider the edges in E_{ij}. As before, we can lower bound the number of these heavy edges covered by P with

$$\sum_{e\in E_{ij}}(\sum_{k\in e} X_k - \sum_{k,l\in e} X_kX_l) = \sum_{e\in E_{ij}}\sum_{k\in e} X_k - \sum_{e\in E_{ij}}\sum_{k,l\in e} X_kX_l.$$

We will use $f_3(X)$ to denote the first term and $f_4(X)$ to denote the second. Also, similar to before, we show that, with high probability, $f_3(X)$ is large and $f_4(X)$ small. To bound $f_3(X)$,

we use Chebyshev's inequality. To compute $\mathrm{Var}(f_3(X))$, we rewrite $f_3(X)$ as $\sum_{k\in V_i} d_k X_k$, where d_k is the degree of vertex k in subhypergraph $H_{ij} = (V_i, E_{ij})$. Thus,

$$\mathrm{Var}(f_3(X)) = \sum_{k\in V_i} d_k^2 \mathrm{Var}(X_k) \leq \frac{\delta}{(1+\epsilon)^j} \sum_{k\in V_i} d_k^2.$$

Since no vertex covers a $\delta^3/(1+\epsilon)$ fraction of E_{ij}, $d_k \leq \delta^3|E_{ij}|$ for all $k \in V_i$. Thus,

$$\begin{aligned}\mathrm{Var}(f_3(X)) &\leq \frac{\delta}{(1+\epsilon)^j}\delta^3|E_{ij}| \sum_{k\in V_i} d_k \\ &\leq \frac{\delta^4|E_{ij}|}{(1+\epsilon)^j}|E_{ij}|(1+\epsilon)^j \\ &\leq \delta^4|E_{ij}|^2.\end{aligned}$$

Also, $E[f_3(X)] \geq |E_{ij}|\delta/(1+\epsilon)$. Therefore, with probability at least 7/8,

$$(f_3(X) - E[f_3(X)])^2 \leq 8\mathrm{Var}(f_3(X)) \leq 8\delta^4|E_{ij}|^2$$

which implies $f_3(X) \geq |E_{ij}|(\frac{\delta}{1+\epsilon} - \sqrt{8}\delta^2)$. The expected value of $f_4(X)$ is at most $|E_{ij}|\delta^2/2$. So with probability at least 3/4,

$$f_4(X) \leq 4E[f_4(X)] \leq |E_{ij}|2\delta^2.$$

With probability at least 5/8 both conditions hold, and thus the number of edges of E_{ij} covered is at least

$$|E_{ij}|(\frac{\delta}{1+\epsilon} - \sqrt{8}\delta^2 - 2\delta^2) \geq |E_{ij}|\delta(1-\epsilon-5\delta) \geq \frac{\delta}{2}|E_{ij}|.$$

With probability at least 1/8, all four conditions hold simultaneously, and P is good. □

With the previous lemma in place, we can now analyze the running time of our *RNC* algorithm.

Lemma 1.2.5 *Without loss of generality, we can assume that for all $v \in V$, $1 \leq c(v) \leq mn^2/\epsilon$.*

Proof Let $\hat{c} = \max_{e\in E} \min_{v\in e} c(v)$. This is a lower bound on the cost of any cover. Therefore, we can include in the cover all vertices that cost less than $\epsilon\hat{c}/n^2$, and only increase the cost of the cover by a factor of $1 + \frac{\epsilon}{n}$. Moreover, since $\hat{c}$ is an upper bound on the cost of covering any edge, we can ignore all vertices that cost more than $m\hat{c}$. Finally, we can multiply all costs by $n^2/\epsilon\hat{c}$ to put all costs in the desired range. Note that all this can easily be done in logarithmic time with a linear number of processors. □

Corollary 1.2.6 *The number of selection steps is* $O(\log^2 n \log m \log(\frac{nm}{\epsilon})/\epsilon^3\delta^3)$.

Proof It follows from the preceding lemma that the number of stages is $O(\log(\frac{mn}{\epsilon})/\epsilon)$. Within each stage, there are $O(\log m/\epsilon)$ phases. Within each phase, there are $O(\log n/\epsilon)$ subphases. And within each subphase, each selection step removes at least a $\delta^3/(1+\epsilon)$ fraction of P_j. Thus, there are $O(\log n/\delta^3)$ selection steps within each subphase, and $O(\log^2 n \log m \log(\frac{nm}{\epsilon})/\epsilon^3\delta^3)$ overall. □

Theorem 1.2.7 *For* $0 < \epsilon < 1$, *there is an* RNC *algorithm for weighted set cover which uses a linear number of processors (i.e.* $O(\sum_{e\in E}|e| + n)$ *processors), runs in expected* $O(\log^2 n \log m \log^2(\frac{nm}{\epsilon})/\epsilon^6)$ *time, and produces a cover of weight at most* $(1+\epsilon)(1+\ln\Delta)\tau^*$.

1.3 *NC* Algorithms for Approximate Set Cover

1.3.1 Exhaustive Search

In the previous section, we gave an *RNC* algorithm for the selection procedure which only required pairwise independence. This can be easily converted to an *NC* algorithm by using a derandomization technique first developed by Karp and Wigderson [43], and later extended by Luby [48] and Alon, Babai, and Itai [2].

Given an *RNC* algorithm A which outputs either a solution or "failure," and is guaranteed to perform well (i.e. succeeds with non-zero probability) under any pairwise independent distribution, we proceed as follows. First we construct a polynomial-sized, pairwise independent distribution $\mathcal{D}$. Since algorithm A succeeds on $\mathcal{D}$ with non-zero probability, there must be some sample point X of $\mathcal{D}$ on which the algorithm succeeds. To find X deterministically, we can run algorithm A on every sample point of $\mathcal{D}$ checking whether or not it succeeds. This is guaranteed to find a good X, and the output of our algorithm is the output of A on random input X. The running time of our deterministic algorithm does not increase (more than an additive $O(\log n)$) from the running time of A. However, the number of processors will increase by a factor equal to the number of sample points our distribution. As long as this is polynomial, we will have an *NC* algorithm.

For our *RNC* set cover algorithm of the previous section, we need pairwise independent random variables $X_1, \ldots, X_n$, which are 1 with probability $\delta/(1+\epsilon)^j$, and 0 otherwise. To

generate such a distribution, we first generate a pairwise independent distribution on $GF(q)$, where $q \geq n$ is a prime power. To do this, we pick a, b at random from $GF(q)$, and let

$$Y_i = af_i + b,$$

where $f_1, \ldots, f_n$ are distinct elements of $GF(q)$. First, we note that the Y_i's are uniformly distributed over $GF(q)$. This is because, for any index i and any $a, r \in GF(q)$, there is exactly one $b \in GF(q)$ which makes $Y_i = x$ (namely $b = r - af_i$). To see that the Y_i's are pairwise independent, we note that for any i and j, $1 \leq i < j \leq n$, and $r, s \in GF(q)$,

$$\begin{aligned} \Pr[Y_i = r \wedge Y_j = s] &= \Pr[af_i + b = r \wedge af_j + b = s] \\ &= 1/q^2. \end{aligned}$$

The last equality comes from polynomial interpolation: there is exactly one polynomial $p(x) = ax + b$ such that $p(f_i) = r$ and $p(f_j) = s$ (namely $a = (r - s)/(f_i - f_j)$ and $b = r - af_i$). Thus we have that

$$\Pr[Y_i = r \wedge Y_j = s] = \Pr[Y_i = r] \Pr[Y_j = s],$$

thus the Y_i's are pairwise independent.

To get pairwise independent X_i's, we let I be a subset of $GF(q)$ of size $q\delta/(1+\epsilon)^j$, and let X_i be 1 if $Y_i \in I$ and 0 otherwise. The pairwise independence of the X_i's follows directly from the pairwise independence of the Y_i's.

The size of the distribution we just created is at most q^2 (corresponding to all possible values for a and b), which can be made to be $O(n^2)$. By exhaustive search and by use of the fact that the *RNC* algorithm of the previous section required only pairwise independence, we get a *NC* algorithm for approximate set cover.

Theorem 1.3.1 *For $0 < \epsilon < 1$, there is an* NC *algorithm for weighted set cover which uses $O(n^2(\sum_{e \in E} |e| + n))$ processors, runs in expected $O(\log^2 n \log m \log^2(nm)/\epsilon^6)$ time, and produces a cover of weight at most $(1+\epsilon)(1 + \ln \Delta)r^*$.*

1.3.2 Binary Search

In the previous section, we saw that if we had a *RNC* algorithm for a problem which only depended on pairwise independence, we could convert this to an *NC* algorithm by exhaustively

searching a small pairwise independent distribution for a good sample point. However, this exhaustive search entailed an increase in the number of processors used.

In this section, we will show how to use a more efficient search technique—a form of binary search—to search a pairwise independent distribution for a good sample point. This method was first developed by Raghavan and Spencer [57, 60, 61] to remove randomness from randomized algorithms which used full independence, resulting in deterministic polynomial-time algorithms. Luby [49] applied the technique to parallel algorithms using pairwise independence—the case we will consider here. In Chapter 2, we will extend this technique to work on parallel algorithms using logarithmic independence.

Consider for the moment the problem of searching a list. While exhaustive search can be used with any list, the more efficient binary search only works on a special form of list—sorted lists. When searching a distribution, as with when searching a list, to achieve a more efficient searching algorithm, the problem must have some additional structure.

The structure required to apply the techniques of [49] is as follows. We require that there be a function F, which Luby calls the *benefit function*, with the following properties:

- F is a function from sample points (settings of $X_1, \ldots, X_n$) to real numbers;
- F measures the goodness of a sample point; specifically, it must be the case that if $F(X) \geq E[F(X)]$, then X is a good sample point.
- F can be written as a sum of terms depending on one or two of the X_i's each;

Note that the last two properties taken together imply that pairwise independence is sufficient to obtain a good solution. This is because goodness is ensured by getting X with $F(X) \geq E[F(X)]$ and $F(X)$ has the same expected value under any pairwise independent distribution as under the fully independent distribution.

Let us restrict our attention to the case where $\Pr[X_i = 0] = \Pr[X_i = 1] = 1/2$. We now show how to find an X with $F(X) \geq E[F(X)]$ for any function F which is a sum of terms depending on one or two random variables each.

First we define a pairwise independent distribution for the X_i's. Let $l = \lceil \log(n+1) \rceil$. Pick

$\omega = \omega_1\omega_2\cdots\omega_{l+1}$ at random from $\{0,1\}^{l+1}$, and let

$$X_i = \sum_{k=1}^{l} \omega_k i_k + \omega_{l+1} \bmod 2,$$

where $i_1 i_2 \cdots i_l$ is the binary expansion of i. We claim that the X_i's are uniformly distributed over { 0,1 } and pairwise (actually 3-wise) independent (for a proof, see [49] or Section 2.2.6).

To find a good X, we will set one bit of ω at a time, thereby performing a binary search on the sample space. This is done as follows. At the beginning of iteration t, assume we have set $\omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}$. Then we compute $E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 0]$ and $E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 1]$. We then set ω_t to the $s_t \in \{0,1\}$ which maximizes $E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = s_t]$.

Lemma 1.3.2 *After step t of the above procedure,* $E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t\,] \geq E[F(X)]$.

Proof (by induction on t)
The case $t = 0$ is clearly true. Assume this lemma is true for $t-1$; i.e. we have $E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}\,] \geq E[F(X)]$. Then

$$\begin{aligned}
&E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t\,] \\
&= \max(E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 0],\ E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 1]) \\
&\geq (E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 0] + E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 1])/2 \\
&= E[\,F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}\,] \\
&\geq E[F(X)] \quad \text{(by inductive hypothesis).}\ \square
\end{aligned}$$

Corollary 1.3.3 *The output of the above procedure is an X such that* $F(X) \geq E[F(X)]$.

It remains to show how to compute the conditional expectations called for in the above algorithm. Assume

$$F(X) = \sum_{1 \leq i \leq n} F_i(X_i) + \sum_{1 \leq i < j \leq n} F_{ij}(X_i, X_j).$$

Then, by linearity of expected value,

$$\begin{aligned}
&E[F(X) | \omega_1 = s_1, \ldots, \omega_t = s_t] \\
&= \sum_{1 \leq i \leq n} E[F_i(X_i) | \omega_1 = s_1, \ldots, \omega_t = s_t] + \sum_{1 \leq i < j \leq n} E[F_{ij}(X_i, X_j) | \omega_1 = s_1, \ldots, \omega_t = s_t].
\end{aligned}$$

To compute a conditional expectation of the form $E[F_i(X_i)|\omega_1 = s_1, \ldots, \omega_t = s_t]$ we consider two cases. If $t \leq l$, then X_i is equally likely to be 0 or 1, thus the conditional expectation is $(F_i(0) + F_i(1))/2$. Otherwise, $t = l + 1$ and the value of X_i is determined; thus the conditional expectation is F_i applied to that value.

To compute a conditional expectation of the form $E[F_{ij}(X_i, X_j)|\omega_1 = s_1, \ldots, \omega_t = s_t]$ we must consider four cases. If $t = l + 1$, then the values of X_i and X_j are determined; thus the conditional expectation is F_{ij} applied to those values. Let r be the index of the last bit where i and j differ. If $t < r$, then X_i and X_j will take on all 4 possible combinations with equal probability; thus the conditional expectation is $(F_{ij}(0,0) + F_{ij}(0,1) + F_{ij}(1,0) + F_{ij}(1,1))/4$. If $r \leq t \leq l$, then X_i and X_j are individually unknown, but are correlated. Let $s_i = \sum_{k=1}^{t} i_k \omega_k \bmod 2$ and $s_j = \sum_{k=1}^{t} j_k \omega_k \bmod 2$. If $s_i = s_j$, then X_i will equal X_j, so the conditional expectation is $(F_{ij}(0,0) + F_{ij}(1,1))/2$. If $s_i \neq s_j$, then X_i will not equal X_j, so the conditional expectation is $(F_{ij}(0,1) + F_{ij}(1,0))/2$.

To compute the conditional expectation of F, we need one processor for each non-zero term in the expansion of F, $O(1)$ time to compute the conditional expectation of each term, and $O(\log n)$ time to sum up all the conditional expectations. Thus to find an X with $F(X) \geq E[F(X)]$, we need $O(\log^2 n)$ time on m processors, where m is the number of non-zero terms in the expansion of F.

If the X_i's are to be uniformly distributed from $\{0,1\}^q$, we can find an X with $F(X) \geq E[F(X)]$ by using q applications of the 1-bit procedure just described. More specifically, let X_i^j be the j^{th} bit of X_i and let X^j be the collection of j^{th} bits. Then given $X^1, \ldots, X^{j-1}$, let

$$F^j(X^j) = E[F(X)|X^1, \ldots, X^j].$$

We observe that $E[F^1(X^1)] = E[F(X)]$, $E[F^j(X^j)] = F^{j-1}(X^{j-1})$, and $F^q(X^q) = F(X)$. Thus, to find an X such that $F(X) \geq E[F(X)]$, we need only use the 1-bit procedure to find for $j = 1, 2, \ldots, q$, an X^j such that $F^j(X^j) \geq E[F^j(X^j)]$. Thus we can find a good X in $O(q \log^2 n)$ steps on m processors. More details of this approach appear in [49] as well as in Section 2.4.2.

Now we show how to apply the above techniques to our *RNC* set cover algorithm. To see that we can put our selection step in the necessary framework, observe that in the proof of

Lemma 1.2.4 we actually showed that the following four conditions imply that P is good:

$$\begin{aligned}
(f_1(X) - E[f_1(X)])^2 &\leq 4\mathrm{Var}(f_1(X)), \\
f_2(X) &\leq 4E[f_2(X)], \\
(f_3(X) - E[f_3(X)])^2 &\leq 8\mathrm{Var}(f_3(X)), \text{ and} \\
f_4(X) &\leq 4E[f_4(X)],
\end{aligned}$$

where

$$\begin{aligned}
f_1(X) &= \sum_{k \in V_i} X_k, \\
f_2(X) &= \sum_{e \in E_i} \sum_{k,l \in e} X_k X_l, \\
f_3(X) &= \sum_{e \in E_{i,j}} \sum_{k \in e} X_k, \text{ and} \\
f_4(X) &= \sum_{e \in E_{i,j}} \sum_{k,l \in e} X_k X_l.
\end{aligned}$$

We can capture these four conditions in a benefit function as follows. Let $F(X)$ be

$$1 - \frac{(f_1(X) - E[f_1(X)])^2}{4\mathrm{Var}(f_1(X))} - \frac{f_2(X)}{4E[f_2(X)]} - \frac{(f_3(X) - E[f_3(X)])^2}{8\mathrm{Var}(f_3(X))} - \frac{f_4(X)}{4E[f_4(X)]}.$$

Clearly, $E[F(X)] = 1/8$. It is also clear that if $F(X) \geq 0$, then the four conditions above are satisfied. Therefore we can apply the techniques of [49] to get a good P. The number of processors used is $O(\sum_{e \in E_i} |e|^2 + |V_i|^2)$, which is at most n times the input size of $O(\sum_{e \in E} |e| + n)$. The running time for one selection step is at most $\log^2 n \log(nm)$. This gives us the following theorem

Theorem 1.3.4 *For $0 < \epsilon < 1$, there is an* NC *algorithm for weighted set cover which uses $O(\sum_{e \in E} |e|^2 + n^2)$ processors, runs in $O(\log^4 n \log m \log^2(nm)/\epsilon^6)$ time, and produces a cover of weight at most $(1 + \epsilon)(1 + \ln \Delta)\tau^*$.*

In the next section. we show how to obtain the same result using only a linear number of processors.

1.3.3 Achieving a Linear Number of Processors

In Section 1.2, we presented an *RNC* algorithm for set cover, which depend on only pairwise independence. This algorithm uses only a linear number of processors. However, applying Luby's method to make this algorithm deterministic causes an increase in the number of processors, since we require one processor for each term of the benefit function F, expanded as a sum of functions depending on one or two variables each. The reason the benefit function has too many terms is that it includes sums of all pairs of a subset of the random variables. To achieve a linear number of processors, we adapt a trick used by Luby [49] to obtain a linear-processor maximal independent set algorithm. Instead of computing conditional expectations on the terms of the expanded benefit function, we compute conditional expectations on terms of the form $\sum_{i,j\in S} X_i X_j$ directly, using $O(|S|)$ processors. (Note that we can rewrite terms of the form $(\sum_{i\in S} X_i - E[\sum_{i\in S} X_i])^2$ as twice the sum of all pairs of S plus $O(|S|)$ other one-variable terms.)

We will demonstrate how to compute the conditional expectations in the simple case where the X_i's are unbiased 0-1 valued random variables. This easily generalizes to the case where the X_i's are identically distributed but biased.

We provide a way to compute

$$E[\sum_{i,j\in S} X_i X_j \mid \omega_1 = s_1, \omega_2 = s_2, \ldots, \omega_t = s_t],$$

in $O(\log n)$ time using $O(|S|)$ processors. If $t = l + 1$, then we know all the X_i's, and $\sum_{i,j\in S} X_i X_j = \left(\sum_{i\in S} X_i\right)$. Otherwise, we partition S into sets $S_\alpha = \{i \in S \mid i_{t+1} \cdots i_l = \alpha\}$. We further partition each S_α into $S_{\alpha,0} = \{i \in S_\alpha \mid \sum_{j=1}^{t} i_j \omega_j = 0 \pmod 2\}$ and $S_{\alpha,1} = S_\alpha - S_{\alpha,0}$. Note that given $\omega_1 = s_1, \ldots, \omega_t = s_t$,

1. $Pr[X_i = 0] = Pr[X_i = 1] = 1/2$,
2. if $i \in S_{\alpha,j}$, and $i' \in S_{\alpha,j'}$, then $X_i = X_{i'}$ iff $j = j'$, and
3. if $i \in S_\alpha$ and $i' \in S_{\alpha'}$, where $\alpha \neq \alpha'$, then $Pr[X_i = X_{i'}] = Pr[X_i \neq X_{i'}] = 1/2$.

Therefore, conditioned on $\omega_1 = s_1, \ldots, \omega_t = s_t$,

$$
\begin{aligned}
E[\sum_{i,j\in S} X_iX_j] &= E[\sum_{\alpha}\sum_{i,j\in S_\alpha} X_iX_j + \sum_{\alpha,\alpha'}\sum_{i\in S_\alpha}\sum_{j\in S_{\alpha'}} X_iX_j] \\
&= \sum_{\alpha} E[\sum_{i,j\in S_{\alpha,0}} X_iX_j + \sum_{i,j\in S_{\alpha,1}} X_iX_j + \sum_{i\in S_{\alpha,0}}\sum_{j\in S_{\alpha,1}} X_iX_j] + \sum_{\alpha,\alpha'}\frac{1}{4}|S_\alpha||S_{\alpha'}| \\
&= \sum_{\alpha}\left[\frac{1}{2}\binom{|S_{\alpha,0}|}{2} + \frac{1}{2}\binom{|S_{\alpha,1}|}{2} + 0|S_{\alpha,0}||S_{\alpha,1}|\right] + \frac{1}{2}\sum_{\alpha}\frac{1}{4}|S_\alpha|\sum_{\alpha'\neq\alpha}|S_{\alpha'}| \\
&= \frac{1}{2}\sum_{\alpha}\left[\binom{|S_{\alpha,0}|}{2} + \binom{|S_{\alpha,1}|}{2}\right] + \frac{1}{8}\sum_{\alpha}|S_\alpha|(|S| - |S_\alpha|).
\end{aligned}
$$

Since there are at most $|S|$ non-empty S_α's, we can compute this using $O(|S|)$ processors. This gives us the following theorem.

Theorem 1.3.5 *For $0 < \epsilon < 1$, there is an* NC *algorithm for weighted set cover which uses a linear number of processors (i.e. $O(\sum_{e\in E}|e| + n)$ processors), runs in $O(\log^4 n \log m \log^2(nm)/\epsilon^6)$ time, and produces a cover of weight at most $(1+\epsilon)(1+\ln\Delta)\tau^*$.*

1.4 Application to Learning Theory

In this section, we apply the set cover algorithm of Section 1.3 to parallel learning, a field first explored by Vitter and Lin [63]. In particular, we provide an *NC* algorithm for learning in concept classes that are formed by taking either finite unions or finite intersections of a fixed base class of finite VC dimension. For example, convex polygons are defined by finite intersections of half-planes. We show that classes of this type are *NC*-learnable whenever there is an *NC* algorithm for finding a consistent hypothesis in the base class (Theorem 1.4.10). In obtaining this result, we employ our parallel set cover algorithm to find a sufficiently simple explanation of the sample data. Blumer et al. [18] previously solved this problem in a polynomial-time model.

The only previous parallel work on this subject is in [63]. They give an *RNC* algorithm for learning s-fold unions of axis-parallel rectangles in the plane, by using a randomized set cover algorithm which is heavily tied to their specific problem. Our general techniques apply directly to this problem. In addition, while Vitter and Lin [63, 64] may produce a hypothesis with up to $s^2 \log m$ rectangles, we will always produce one with at most $s \log m$ rectangles, which is within a logarithmic factor of optimal.

The following definitions are adapted from [18, 63]:

Definition 1.4.1 Fix a domain X. A *concept class* is a nonempty set $C \subseteq 2^X$ of *concepts.* In this chapter, it is assumed that X is a fixed set, either finite, countably infinite, $[0,1]^n$, or E^n for some $n \geq 1$. The length of a concept c, denoted $|c|$, is the number of bits required to write c in some standard encoding.

Definition 1.4.2 Given a nonempty concept class $C \subseteq 2^X$ and a set of points $S \subseteq X$, $\Pi_C(S)$ denotes the set of all subsets of S that can be obtained by intersecting S with a concept in C; i.e., $\Pi_C(S) = \{S \cap c \mid c \in C\}$. For any integer $m \geq 0$, $\Pi_C(m) = \max(|\Pi_C(S)|)$ over all $S \subseteq X$ of cardinality m. The *Vapnik-Chervonenkis (VC) dimension* of C is the largest integer d such that $\Pi_C(d) = 2^d$, or ∞ if there is no such d.

Definition 1.4.3 Let C be defined as above. We say that C is *NC-learnable* if there exists an NC algorithm A that takes as input a sample of a concept in C, outputs a hypothesis in C, and has the property that for all $0 < \epsilon, \delta < 1$, and $s \geq 1$ there exists a sample size $m(\epsilon, \delta, s)$, polynomial in $1/\epsilon$, $1/\delta$, and s, such that for all target concepts $c \in C$ with $|c| \leq s$, and all probability distributions P on X, given a random sample of c of size $m(\epsilon, \delta, s)$ drawn independently according to P, the algorithm A produces, with probability at least $1 - \delta$, a hypothesis $h \in C$ that has error at most ϵ, i.e. $Pr_P[X \in c \oplus h] \leq \epsilon$, where $\oplus$ is the symmetric difference.

Definition 1.4.4 Let $C \subseteq 2^X$ be a concept class. By $U(C)$ we denote the closure of C under finite unions, i.e.,

$$U(C) = \{c_1 \cup \cdots \cup c_s \mid s \geq 1 \text{ and } c_i \in C,\ 1 \leq i \leq s\}.$$

Similarly, $I(C)$ denotes the closure of C under finite intersections.

Definition 1.4.5 Let C be a concept class. An *NC hypothesis finder* for C is an NC algorithm that, given a sample of a target concept C, returns a hypothesis in C that is consistent with the sample. Note that we do not consider randomized hypothesis finders here. The *consistency problem for* C is the problem of determining if there is a concept in C that is consistent with a given sample over X. Note that the existence of an NC hypothesis finder for C implies that the consistency problem for C is in NC.

Definition 1.4.6 Let C be a concept class defined on domain X. Let A be an NC algorithm that, given a sample of a concept in C, produces a consistent hypothesis in C. For every $s, m \geq 1$, let $S_{C,s,m}$ denote the set of all m-samples of concepts $c \in C$ such that $|c| \leq s$. Let $C^A_{s,m} \subseteq C$ denote the A-image of $S_{C,s,m}$, i.e., the set of all hypothesis produced by A when A is given as input an m-sample of a concept $c \in C$ with $|c| \leq s$. We will call $C^A_{s,m}$ the *effective hypothesis space of A for target complexity s and sample size m*. We say A is an *NC-Occam algorithm* for C if there exists a polynomial $p(s)$ and a constant α, $0 \leq \alpha < 1$, such that for all $s, m \geq 1$ the VC dimension of $C^A_{s,m}$ is at most $p(s)m^\alpha$.

Theorem 1.4.7 ([18]) *Let C be a concept class with a given concept complexity measure. If there is an* NC*-Occam algorithm for C then C is* NC*-learnable.*

Proposition 1.4.8 ([62, 59]) *If the VC dimension of H is $d \geq 0$, then $\Pi_H(m) \leq m^d + 1$.*

We now use Theorem 1.4.7 to demonstrate the learnability of many concept classes of the form $U(C)$ and $I(C)$ for C of finite VC dimension.

Lemma 1.4.9 *If C has finite VC dimension $d < \infty$ and the consistency problem for C is in* NC*, then for any finite set $S \subseteq X$, the sets of $\Pi_C(S)$ can be listed in* NC.

Proof Assume $S = \{x_1, \ldots, x_m\}$. To produce a list L of $\Pi_C(S)$, we proceed as follows. If $m = 1$, we check if $\emptyset$ and $\{x_1\}$ are consistent and return $\{T \subseteq S \mid T \text{ is consistent}\}$. If $m > 1$, then we recurse (in parallel) to get $L_1 = \Pi_C(\{x_1, \ldots, x_{\lfloor m/2 \rfloor}\})$ and $L_2 = \Pi_C(\{x_{\lfloor m/2 \rfloor + 1}, \ldots, x_m\})$. Then in parallel for all pairs $T_1 \in L_1$, $T_2 \in L_2$, we check the consistency of $T_1 \cup T_2$, and return $\{T_1 \cup T_2 \mid T_1 \in L_1,\ T_2 \in L_2,\ T_1 \cup T_2 \text{ is consistent}\}$. To check the consistency of $T \subseteq S$, we run the NC consistency algorithm for C on a sample consisting of positive examples T and negative examples $S - T$. The depth of the recursion is $\log m$; furthermore, since by Proposition 1.4.8 the size of $\Pi_C(S)$ is at most $|S|^d + 1$, we run the consistency algorithm at most $O(m^{2d})$ times in parallel at each level of the recursion. Therefore, this algorithm is in NC. □

Theorem 1.4.10 *Let C be a concept class with VC dimension $d < \infty$ such that there exists an* NC *hypothesis finder for C. Then $U(C)$ (resp. $I(C)$) is* NC*-learnable.*

Proof We consider only the case $U(C)$, the other case being similar. Let S be the set of points in an m-sample of a target concept c in $U(C)$. Our strategy will be to find a hypothesis consistent with S that is formed from the union of relatively few concepts in C; i.e. not many more than s. This problem can be formulated as a set cover problem. The set to be covered is the set of positive points of S and the sets allowed in the cover are the elements of $\Pi_C(S)$ that contain only positive points.

By Lemma 1.4.9, we can construct $\Pi_C(S)$ in NC. Then, in parallel, we can easily compute $\mathcal{A} = \{T \in \Pi_C(S) \mid T \text{ contains only positive points of } S\}$ and $P = \{x \in S \mid x \text{ is a positive example}\}$. We can then apply the set cover algorithm of Section 1.3 to obtain a cover of size $O(s \log m)$. For each set in the cover, we can label the other points negative and run the NC hypothesis finder for C to produce a hypothesis in C that contains only these points of the sample. Taking the union of these concepts, we obtain a hypothesis in $U(C)$. Call this algorithm A.

We have shown that A is in NC and that given any m-sample of a concept c in $U(C)$ with $|c| \leq s$, A produces a consistent hypothesis h for this sample with $|h| \leq O(s \log m)$. Hence the effective hypothesis space $U(C^A_{s,m})$ of A for target complexity s and sample size m contains only hypotheses such that $|h| \leq O(s \log m)$. It follows from a lemma of Blumer, et al. that the VC dimension of $U(C^A_{s,m})$ is $O(s \log(m)(\log s + \log\log m))$. Hence, A is an NC-Occam algorithm for $U(C)$ and thus by Theorem 1.4.7, $U(C)$ is NC-learnable. □

Chapter 2

Removing Randomness from Parallel Algorithms Using Logarithmic Independence

2.1 Introduction

As we have seen, for many applications [61, 57, 43, 48, 2], the problem of removing randomness from an algorithm can be solved by finding an $X = < X_1, \ldots, X_n >$ such that $F(X) \geq E[F(X)]$, for some benefit function F and some sample space $\mathcal{S}$ over which the expectation is to be computed. The problem is then how best to find a good sample point (e.g., an X such that $F(X) \geq E[F(X)]$) in $\mathcal{S}$. If the space of sample points is small (e.g., polynomial), then this can be accomplished by brute force; namely, we could try all points until we get a good one, for one must exist. We saw this in the previous chapter, where we constructed an *RNC* algorithm for set cover which only depended on pairwise independence, and so we were able to remove the randomness by this method and obtain an *NC* algorithm. However, many applications seem to inherently require more than constant independence, and thus sample spaces which are larger than polynomial, making brute force too expensive. In such situations, the only general method available is the *method of conditional probabilities*, devised by Spencer

This chapter describes joint work with Bonnie Berger [14].

[60, 61] and further developed by Raghavan [57]. This method works by setting the X_i's one by one in such a way as to not decrease the conditional expectation (e.g., setting X_{i+1} so that $E[F(X) \mid X_1, \ldots, X_{i+1}] \geq E[F(X) \mid X_1, \ldots, X_i]$). The main difficulty with this approach is in computing the conditional expectations — the ability to do so determines when the method can and cannot be used.

Unfortunately, the method of conditional probabilities is inherently sequential; hence, the running time of the Spencer and Raghavan algorithms is at least n. Since the best time one could hope for is logarithmic in the size of the sample space, for large sample spaces this is probably as good as one can get; yet, for smaller spaces, n is far from optimal. The only improvement to this approach was by Luby [49] (see also Section 1.3.2) who showed how to search in time logarithmic in the size of the sample space for the special case of pairwise independence, thereby improving the processor efficiency of several *NC* algorithms ($\Delta + 1$ vertex coloring, MIS, and maximal matching) from $n^2(n+m)$ to $n+m$.

In this chapter, we substantially increase the size of sample space we can derandomize in general. We show how to search in time logarithmic in the size of the sample space for a wide range of functions F and arbitrarily large sample spaces. As a result, we can prove substantially stronger results than is possible with the Luby method. In particular, we are able to derive *NC* algorithms for several problems that were not previously known to be in *NC*, and we can search $(\log^c n)$-wise independent $n^{\log^c n}$-size sample spaces in *NC*.

In Section 2.2, we demonstrate how our techniques apply to the problem of set discrepancy. In this problem, we are given a set of n points and n subsets of at most Δ of these points, and we want to color the points 0 and 1 so that the discrepancy, or maximum difference between the number of 0's and the number of 1's in any subset, is small. The best known randomized (parallel) algorithm achieves a discrepancy of $O(\sqrt{\Delta \log n})$; Spencer [60] applied the method of conditional probabilities to this algorithm to obtain a deterministic sequential algorithm with the same bound. We show that the randomized algorithm requires $\frac{\log n}{\epsilon \log \Delta}$-independence to give a discrepancy bound of $\Delta^{1/2+\epsilon}\sqrt{\log n}$ for any fixed $\epsilon > 0$; then we apply our techniques to convert this to an *NC* algorithm which attains the same bound (using $n^{1/\epsilon}$ processors). This algorithm has many applications. As an example, we give a deterministic version of the Karloff-Shmoys parallel edge coloring algorithm [41], obtaining the same $\Delta + \Delta^{1/2+\epsilon}$ bound on the number of

colors used as their randomized algorithm and beating the known deterministic bound of $2\Delta-1$ [49]. The results of Section 2.2 first appeared in [13]. Results similar to those in Section 2.2 were subsequently discovered by Motwani, Naor, and Naor [53], and further work along these lines appears in [54].

In Section 2.3, we show how to apply our techniques to a large class of problems which depend on $(\log^c n)$-wise independence. In particular, we describe an *NC* algorithm for obtaining the expected value of any function which is the sum of a polynomial number of terms, each depending on $O(\log n)$ binary random variables; e.g., a function of the form

$$F(X) = \sum_{i=1}^{n^a} f_i(X_{i,1}, \ldots, X_{i,b\log n}).$$

Alternatively, we can allow the terms to be simple functions of $\log^c n$ random variables; e.g., characteristic functions of affine subspaces, which by a reduction can be used to build any function which is non-zero for only a polynomial number of points.

In Section 2.4, we give several methods for extending our technique to multivalued random variables. As an illustration, we consider the hypergraph coloring problem: given a d-uniform hypergraph $(V, \mathcal{E})$, color the vertices with d colors so that at least $d!|\mathcal{E}|/d^d$ edges have one vertex of each color. If d is a constant, this problem can be solved by trying all sample points in a d-wise independent distribution [2]. Using our techniques for handling $(\log n)$-wise independent multivalued random variables, we give a deterministic *NC* algorithm which solves this problem for all d. The particular technique used to handle the multivalued random variables in this problem involves generating and solving a series of problems with binary random variables, highlighting the importance of being able to solve a large class of these.

Finally, in Section 2.5, we provide improved algorithms and bounds for set discrepancy. Among the results in this section is an *NC* algorithm for weighted discrepancy, which has applications to other problems such as lattice approximation. Also included is an *NC* algorithm which achieves an $O(\sqrt{\Delta \log n})$ discrepancy bound for the case when Δ is polylogarithmic.

2.2 An Example of the Method — Set Discrepancy

2.2.1 Definition of Problem

Spencer [61, p. 30] defines the *set discrepancy* problem as follows. Let $\mathcal{A} \subseteq 2^{\Gamma}$, $|\mathcal{A}| = |\Gamma| = n$, be a family of finite sets. Let $\chi : \Gamma \rightarrow \{-1, +1\}$ be a 2-coloring of the underlying points. Define

$$\begin{aligned} \chi(A) &= \sum_{i \in A} \chi(i) \\ disc(\chi) &= \max_{A \in \mathcal{A}} |\chi(A)|. \end{aligned}$$

We want to find a χ such that $disc(\chi)$ is small.

How small can we make $disc(\chi)$? Spencer [61, p. 73-77] shows that there exists a χ with $disc(\chi) = O(\sqrt{n})$. He also shows that this is the best possible; i.e. that there exists an $\mathcal{A}$ such that all χ have $disc(\chi) = \Omega(\sqrt{n})$.

It is interesting to bound discrepancy in terms of maximum degree $\Delta = \max_{A \in \mathcal{A}} |A|$. Spencer's lower bound can be easily modified to give, for any Δ, an $\mathcal{A}$ with cardinality n and maximum degree Δ such that all χ have $disc(\chi) = \Omega(\sqrt{\Delta})$. It is easily shown in Section 2.2.2 that if we pick χ at random, with high probability $disc(\chi)$ is at most $2\sqrt{\Delta \log n}$. This immediately gives an *RNC* algorithm achieving that bound. Spencer [60] shows how to convert this into a deterministic poly-time algorithm. In the sections to follow, we develop an *NC* algorithm which outputs a χ with $disc(\chi) \leq \Delta^{1/2+\epsilon}\sqrt{\log n}$.

An interesting special case of the set discrepancy problem is the *graph discrepancy* problem. Given a graph $G = (V, E)$, we want to find a 2-coloring of the vertices $\chi : V \rightarrow \{-1, +1\}$ such that $\max_{v \in V} |\sum_{u \in N(v)} \chi(u)|$ is small, where $N(v) = \{u | (v, u) \in E\}$. We can reduce this problem to set discrepancy by letting $\Gamma = V$ and $\mathcal{A} = \{N(v) \,|\, v \in V\}$. Plugging in our *NC* algorithm for discrepancy, we get a χ with $\max_{v \in V} |\sum_{u \in N(v)} \chi(u)| \leq \Delta^{1/2+\epsilon}\sqrt{\log n}$, where Δ is the maximum degree of G.

A variation on the set discrepancy problem is the *weighted discrepancy* problem, where each element of Γ is assigned a real weight between -1 and 1. Then $\chi(A)$ becomes a weighted sum of the $\chi(i)$'s. In Section 2.5.2 we give an *NC* algorithm for this problem.

2.2.2 An *RNC* Algorithm

Consider the following algorithm for set discrepancy: randomly pick χ until one is found such that $disc(\chi) \leq 2\sqrt{\Delta \ln n}$. One iteration of this is clearly in *RNC*. We will show that the expected number of iterations is less than two. The following will prove useful.

Proposition 2.2.1 *[35, 61] Let $X_1, \ldots, X_\Delta$ be independent random variables each with mean 0 and taking on values in the range $[-1, +1]$. Let $S = \sum_i X_i$. Then $Pr[S > \lambda] < e^{-\lambda^2/2\Delta}$.*

Lemma 2.2.2 $Pr\left[disc(\chi) > 2\sqrt{\Delta \ln n}\right] < 2/n$.

Proof For each $A \in \mathcal{A}$, Proposition 2.2.1 shows

$$\begin{aligned} Pr\left[|\chi(A)| > 2\sqrt{\Delta \ln n}\right] &\leq 2Pr\left[\chi(A) > 2\sqrt{|A| \ln n}\right] \\ &< 2e^{-\left(2\sqrt{|A| \ln n}\right)^2/2|A|} \\ &= 2/n^2. \end{aligned}$$

Thus,

$$\begin{aligned} Pr\left[disc(\chi) > 2\sqrt{\Delta \ln n}\right] &\leq \sum_{A \in \mathcal{A}} Pr\left[|\chi(A)| > 2\sqrt{\Delta \ln n}\right] \\ &\leq 2/n. \quad \square \end{aligned}$$

Thus, the expected number of iterations is at most $1/\left(1 - \frac{2}{n}\right) < 2$. So the above is clearly an *RNC* algorithm. Also, one can easily show using Lemma 2.2.2 that $E[disc(\chi)] \leq 2\sqrt{\Delta \ln n}$.

2.2.3 The Overall Approach

Lemma 2.2.2 shows that the probability of $disc(\chi)$ being larger than $2\sqrt{\Delta \ln n}$ is small. This implies that there exists a χ with $disc(\chi) \leq 2\sqrt{\Delta \ln n}$. We wish to find such a χ deterministically. Unfortunately, the random construction of Section 2.2.2 assumed a fully independent distribution, which must have 2^n sample points. Clearly, we cannot search this sample space exhaustively. However, Spencer [61] developed a method to perform a binary search on the sample space. While he achieves a polynomial time algorithm, it requires n decisions to be

made. Since each decision depends on previous ones, it seems very unlikely that these decisions could be made in parallel. To get an efficient parallel algorithm, we must work with a smaller sample space. A natural choice would be to choose the vector $<\chi(1),\ldots,\chi(n)>$ from a k-wise independent distribution, where k is small.

Ideally our goal is to find a χ with small discrepancy by finding a χ for which $disc(\chi) \leq E[disc(\chi)]$, where the expectation is taken over a k-wise independent distribution for some small k. The choice of k is influenced by two factors:

1. if k is too small, then $E[disc(\chi)]$ might be too large and
2. if k is too large, then finding a χ which achieves the expectation takes too long.

As a compromise, we will eventually choose $k = \frac{\log n}{\epsilon \log \Delta}$, $\epsilon > 0$.

There are other problems with this approach, however. Most important, to find a good χ, we will need to compute expectations of $disc(\chi)$ conditioned on some knowledge of the distribution, in NC. This is hopelessly complicated by the *max* and absolute value in $disc(\chi)$. To get around these problems, we will use higher moments, using $\sum_{A\in\mathcal{A}} |\chi(A)|^k$ as an upper bound on $disc^k(\chi)$. If k is even, this gets rid of both the *max* and the absolute value. In other words, we will

1. show that $E\left[\sum_{A\in\mathcal{A}} |\chi(A)|^k\right]$ is small for suitable k where the expectation is taken over a k-wise independent distribution, and then
2. find a χ such that $\sum_{A\in\mathcal{A}} |\chi(A)|^k \leq E\left[\sum_{A\in\mathcal{A}} |\chi(A)|^k\right]$.

As a consequence, we will have found a χ for which $disc^k(\chi)$ is small, and thus for which $disc(\chi)$ is small. By choosing $k = \frac{\log n}{\epsilon \log \Delta}$, we will produce a χ for which $disc(\chi) \leq \Delta^{1/2+\epsilon}\sqrt{\log n}$. This is not quite the $\sqrt{\Delta \log n}$ bound we got with the RNC algorithm, but it is close.

2.2.4 k-wise Independence Inequalities

In this section we will bound the k^{th} central moment of a sum of k-wise independent bounded random variables. We will need the bounds we generate in this section throughout the thesis, including for the discrepancy bound in this chapter. We will resume discussion of lesser independence and the set discrepancy problem in the next section.

Let $X_1, \ldots, X_n$ be random variables such that X_i takes on real values between $E[X_i]-1$ and $E[X_i]+1$ (e.g. X_i takes on values in the interval $[0,1]$ or X_i takes on values -1 and $+1$ each with probability $1/2$). Let $X = \sum_{i=1}^{n} X_i$ and let $\mu = E[X]$. We will bound the k^{th} central moment, $E[(x-\mu)^k]$. The following lemma implies that it is sufficient to compute this expectation under the assumption that the X_i's are fully independent.

Lemma 2.2.3 *Any function which can be represented as a sum of functions depending on at most k random variables each has the same expected value taken over any distribution with k-wise or greater independence.*

Proof Follows directly from the definition of k-wise independence and linearity of expected value. □

Given fully independent X_i's, we first calculate upper bounds on the probability that X has large deviation from μ. We will then proceed to use these bounds to derive bounds on the k^{th} moment. The following is a simple corollary of Proposition 2.2.1 which we give for convenience.

Lemma 2.2.4 *Suppose $X_1, \ldots, X_n$ are independent random variables such that X_i takes on real values between $E[X_i]-1$ and $E[X_i]+1$, $X = \sum_{i=1}^{n} X_i$, $\mu = E[X]$, and $A > 0$. Then,*

$$\Pr[|X - \mu| > A] < 2e^{-A^2/2n}.$$

Proof Let $Y_i = X_i - E[X_i]$, let $Y = \sum_{i=1}^{n} Y_i = X - \mu$, and apply Proposition 2.2.1. □

Now we proceed to use the above lemma to obtain bounds on the k^{th} central moment, $E[(X-\mu)^k]$. We will assume k is even, and thus $(X-\mu)^k \geq 0$. The following well-known lemma will prove useful:

Lemma 2.2.5 *Let Z be a non-negative real valued random variable. Then,*

$$E[Z] = \int_0^\infty \Pr[Z > x]dx.$$

So it suffices to calculate an upper bound on

$$\int_0^\infty \Pr[(X-\mu)^k > x]dx = \int_0^\infty \Pr[|X-\mu| > x^{1/k}]dx$$

The following integral will prove useful: if c is a non-negative integer, then

$$\int_0^\infty x^c e^{-x} dx = c!.$$

The following form of Stirling's formula which gives an upper bound for all n rather than a limit will also prove useful:

$$n! < e^{1/12n}\sqrt{2\pi n} n^n e^{-n}.$$

We can now prove our upper bound on the k^{th} central moment.

Lemma 2.2.6 *Suppose k is a positive even integer, $X_1, \ldots, X_n$ are k-wise independent random variables such that X_i takes on real values between $E[X_i]-1$ and $E[X_i]+1$, $X = \sum_{i=1}^n X_i$, and $\mu = E[X]$. Then,*

$$E[(X-\mu)^k] \leq 2e^{1/6k}\sqrt{\pi k}\left(\frac{nk}{e}\right)^{k/2}.$$

Proof As mentioned above, it is sufficient to prove the bound under the assumption that the X_i's are fully independent.

$$\begin{aligned} E[(X-\mu)^k] &= \int_0^\infty \Pr[|X-\mu| > x^{1/k}]dx \\ &\leq \int_0^\infty 2e^{-x^{2/k}/2n} dx. \end{aligned}$$

With the change of variables $y = x^{2/k}/2n$, this integral becomes

$$\begin{aligned} & 2\frac{k}{2}(2n)^{k/2}\int_0^\infty y^{k/2-1}e^{-y}dy \\ &= 2\frac{k}{2}(2n)^{k/2}\left(\frac{k}{2}-1\right)! \\ &= 2\left(\frac{k}{2}\right)!(2n)^{k/2} \\ &\leq 2\sqrt{\pi k}e^{1/6k}\left(\frac{k}{2e}\right)^{k/2}(2n)^{k/2} \\ &= 2e^{1/6k}\sqrt{\pi k}\left(\frac{nk}{e}\right)^{k/2}. \quad \square \end{aligned}$$

We can use our bound on the k^{th} moment to obtain an upper bound on the probability of a large deviation of X from its mean, μ.

Lemma 2.2.7 *Suppose k is a positive even integer, $X_1, \ldots, X_n$ are k-wise independent random variables such that X_i takes on real values between $E[X_i]-1$ and $E[X_i]+1$, $X = \sum_{i=1}^{n} X_i$, $\mu = E[X]$, and $A > 0$. Then,*

$$\Pr[|X - \mu| > A] < 2e^{1/6k}\sqrt{\pi k}\left(\frac{nk}{eA^2}\right)^{k/2}.$$

Proof Using Markov's inequality, we see that

$$\begin{aligned}\Pr[|X-\mu| > A] &= \Pr[(X-\mu)^k > A^k] \\ &\leq E[(X-\mu)^k]/A^k \\ &\leq 2e^{1/6k}\sqrt{\pi k}\left(\frac{nk}{eA^2}\right)^{k/2}. \quad \square\end{aligned}$$

Remark If $k \geq 4$, then the above bound has a simpler form, namely

$$\Pr[|X - \mu| > A] \leq \left(\frac{nk}{A^2}\right)^{k/2}.$$

We now have fairly tight results on the behavior of the sum of n k-wise independent random variables. However, if the random variables take on values in the range $[0,1]$, and μ is small, these bounds can be improved. The bounds given above were phrased in terms of k and n; they can actually be stated in terms of k and μ. To show this, we will give versions of Lemmas 2.2.4, 2.2.6, and 2.2.7, phrased in terms of μ rather than n. Although we will not use these bounds in this chapter, we will use them in Chapter 4.

We begin by proving a stronger version of Lemma 2.2.4. Although (stronger) versions of this lemma appear in the literature [35], the form below is simpler and more convenient for our purposes. Also, similar looking bounds which are, in fact, incorrect have also appeared [61].

Lemma 2.2.8 *Suppose $X_1, \ldots, X_n$ are independent random variables taking on values in the range $[0,1]$, $X = \sum_{i=1}^{n} X_i$, $\mu = E[X]$, and $A > 0$. Then,*

$$\begin{aligned}\Pr[|X-\mu| > A] &< \max(2e^{-3A^2/8\mu}, e^{-2A/5}) \\ &< 2e^{-3A^2/8\mu} + e^{-2A/5}.\end{aligned}$$

Proof As in Lemma 2.2.4, let $Y_i = X_i - E[X_i]$ and let $Y = \sum_{i=1}^{n} Y_i = X - \mu$. Also, let $p_i = E[X_i]$. Then, for any t,

$$\begin{aligned} \Pr[X - \mu > A] &\leq \frac{E[e^{tY}]}{e^{tA}} \\ &= \frac{\prod_{i=1}^{n} E[e^{tY_i}]}{e^{tA}}. \end{aligned}$$

By the convexity of the exponential function, we get that

$$E[e^{tY_i}] \leq (1 - p_i)e^{-tp_i} + p_i e^{t(1-p_i)}.$$

By taking Taylor expansions and combining terms, we have

$$\begin{aligned} E[e^{tY_i}] &\leq 1 + p_i(1 - p_i)\left(\frac{t^2}{2!} + ((1-p_i)^2 - p_i^2)\frac{t^3}{3!} + ((1-p_i)^3 + p_i^3)\frac{t^4}{4!} + ((1-p_i)^4 - p_i^4)\frac{t^5}{5!} + \cdots\right) \\ &\leq 1 + p_i\left(\frac{t^2}{2!} + \frac{|t|^3}{3!} + \frac{t^4}{4!} + \frac{|t|^5}{5!}\cdots\right) \\ &= 1 + p_i(e^{|t|} - 1 - |t|) \\ &= 1 + p_i\frac{t^2}{2!}\left(\frac{e^{|t|} - 1 - |t|}{t^2/2}\right). \end{aligned}$$

If we restrict to the case $|t| < 4/5$, we get that

$$E[e^{tY_i}] \leq 1 + p_i\frac{2t^2}{3} \leq e^{2p_i t^2/3},$$

which implies that

$$E[e^{tY}] \leq e^{2\mu t^2/3}.$$

Therefore,

$$\Pr[Y > A] < e^{2\mu t^2/3 - tA}.$$

The optimal value for t in the above formula is $3A/4\mu$. But we must have $|t| \leq 4/5$, so we let $t = \min(3A/4\mu, 4/5)$. For $A \leq 16\mu/15$, $t = 3A/4\mu$, so

$$\Pr[Y > A] < e^{3A^2/8\mu - 3A^2/4\mu} = e^{-3A^2/8\mu}.$$

For $A \geq 16\mu/15$, $t = 4/5$, so

$$\Pr[Y > A] < e^{32\mu/75 - 4A/5} \leq e^{2A/5 - 4A/5} = e^{-2A/5}.$$

Similarly, we note that

$$\Pr[Y < -A] < E[e^{tY}]/e^{-tA},$$

which we can optimize by letting $t = -3A/4\mu$, obtaining

$$\Pr[Y < -A] < e^{-3A^2/8\mu}.$$

Note we do not have to consider the case $A > 16\mu/15$, since $\Pr[Y < -\mu] = 0$. Therefore we have

$$\Pr[|Y| > A] < \max(2e^{-3A^2/8\mu}, e^{-2A/5}),$$

thus proving the lemma. □

Now we proceed to use the above lemma to obtain bounds on the k^{th} central moment, $E[(X-\mu)^k]$, for positive even integers k.

Lemma 2.2.9 *Suppose k is a positive even integer, $X_1, \ldots, X_n$ are k-wise independent random variables taking on values in the range $[0,1]$, $X = \sum_{i=1}^{n} X_i$, and $\mu = E[X]$. Then,*

$$E[(X-\mu)^k] \leq O\left((k\mu + k^2)^{k/2}\right).$$

Proof As in Lemma 2.2.6, by Lemma 2.2.3, it is sufficient to prove the bound under the assumption that the X_i's are fully independent.

$$\begin{aligned} E[(X-\mu)^k] &= \int_0^\infty \Pr[|X-\mu| > x^{1/k}]dx \\ &\leq \int_0^\infty (2e^{-3x^{2/k}/8\mu} + e^{-2x^{1/k}/5})dx \\ &= 2\int_0^\infty e^{-3x^{2/k}/8\mu}dx + \int_0^\infty e^{-2x^{1/k}/5}dx. \end{aligned}$$

With the changes of variables $y = 3x^{2/k}/8\mu$ and $z = 2x^{1/k}/5$, this integral becomes

$$\begin{aligned} & 2\left(\frac{8\mu}{3}\right)^{k/2}\frac{k}{2}\int_0^\infty y^{k/2-1}e^{-y}dy + \left(\frac{5}{2}\right)^k k\int_0^\infty z^{k-1}e^{-z}dz \\ &= 2\left(\frac{8\mu}{3}\right)^{k/2}\frac{k}{2}\left(\frac{k}{2}-1\right)! + \left(\frac{5}{2}\right)^k k(k-1)! \\ &= 2\left(\frac{8\mu}{3}\right)^{k/2}\left(\frac{k}{2}\right)! + \left(\frac{5}{2}\right)^k k! \end{aligned}$$

$$\begin{aligned} &\leq 2\sqrt{\pi k}e^{1/6k}\left(\frac{4k\mu}{3e}\right)^{k/2} + \sqrt{2\pi k}e^{1/12k}\left(\frac{5k}{2e}\right)^{k} \\ &= O\left((k\mu)^{k/2} + (k^2)^{k/2}\right) \\ &\leq O\left((k\mu + k^2)^{k/2}\right). \quad \square \end{aligned}$$

Finally, we use our bound on the k^{th} moment to obtain an upper bound on the probability of a large deviation of X from its mean, μ.

Lemma 2.2.10 *Suppose k is a positive even integer, $X_1, \ldots, X_n$ are k-wise independent random variables taking on values in the range $[0,1]$, $X = \sum_{i=1}^{n} X_i$, $\mu = E[X]$, and $A > 0$. Then,*

$$\Pr[|X - \mu| > A] < O\left(\left(\frac{k\mu + k^2}{A^2}\right)^{k/2}\right).$$

Proof Follows from Lemma 2.2.9 and Markov's inequality. $\square$

2.2.5 Bounding the Independence Needed

Recall that we want to bound $disc^k(\chi)$ by $\sum_{A\in\mathcal{A}} \chi^k(A)$.

Lemma 2.2.11 *Suppose k is a positive even integer. Then, for any k-wise independent distribution,*

$$E[\sum_{A\in\mathcal{A}} \chi^k(A)] \leq n\,O((k\Delta)^{k/2}).$$

Proof For each $A \in \mathcal{A}$, $\chi(A)$ is a sum of Δ random variables taking values -1 and $+1$ equally likely. Thus, by Lemma 2.2.4, for any $A \in \mathcal{A}$,

$$E[\chi^k(A)] \leq O((k\Delta)^{k/2}).$$

The lemma follows from linearity of expectation. $\square$

We can now give a lower bound on the value for k. We want $E[\sum_{A\in\mathcal{A}} \chi^k(A)]^{1/k} \leq \Delta^{1/2+\epsilon}\sqrt{\log n}$; this is roughly captured by having $n^{1/k} \leq \Delta^{\epsilon}$. This implies we need $k = 2\left\lceil \frac{\log n}{2\epsilon \log \Delta} \right\rceil$. If k is thus, and we are able to find a χ such that $\sum_{A\in\mathcal{A}} \chi^k(A)$ is at most its expectation, this implies that

$$disc^k(\chi) \leq \sum_{A\in\mathcal{A}} \chi^k(A) \leq E[\sum_{A\in\mathcal{A}} \chi^k(A)] \leq O(n(k\Delta)^{k/2}).$$

So,

$$\begin{aligned} disc(\chi) &\leq O(n^{1/k}\sqrt{k\Delta}) \\ &\leq O(\Delta^{\epsilon}\sqrt{k}\,\sqrt{\Delta}) \\ &\leq \Delta^{1/2+\epsilon}\sqrt{\log n}. \end{aligned}$$

It is worth pointing out that the preceding analysis is in some sense tight; i.e. that to get $disc(\chi) \leq \Delta^{1/2+\epsilon}$, any method based on independence alone requires at least $\left(\frac{2\log n}{\log \Delta}\right)$-wise independence. This notion is captured by the following theorem.

Theorem 2.2.12 *For any n and Δ, we can construct a $\left(\frac{2\log n}{\log \Delta}\right)$-wise independent distribution and a set system $\mathcal{A}$ of size n and maximum degree Δ such that $E[disc(\chi)] = \Omega(\Delta)$.*

Proof Our distribution is as follows: first pick $\chi(1), \ldots, \chi(\Delta)$ from a $\left(\frac{2\log n}{\log \Delta}\right)$-wise independent distribution with at most n sample points (see [2] for construction). Then, to pad out the distribution to n variables, pick $\chi(\Delta+1), \ldots, \chi(n)$, independently of $\chi(1), \ldots, \chi(\Delta)$, from an arbitrary $\left(\frac{2\log n}{\log \Delta}\right)$-wise independent distribution. Now we construct $\mathcal{A}$ such that for every possible sample point in the distribution for the first Δ variables, there is some set with large discrepancy. This implies that the expected discrepancy is large. $\mathcal{A}$ is constructed as follows: for each sample point in the distribution for $\chi(1), \ldots, \chi(\Delta)$, we include in $\mathcal{A}$ the larger of sets $\{i \mid 1 \leq i \leq \Delta,\ \chi(i) = -1\}$ and $\{i \mid 1 \leq i \leq \Delta,\ \chi(i) = +1\}$. This ensures that for each sample point, we will have a corresponding set $A \in \mathcal{A}$ (with $\Delta/2 \leq |A| \leq \Delta$) whose elements are assigned either all $+1$'s or all -1's. Thus, $disc(\chi) \geq \Delta/2$, which implies $E[disc(\chi)] \geq \Delta/2$. □

The next three sections will be devoted to finding a χ such that $\sum_{A\in\mathcal{A}} \chi^k(A)$ is at most its expectation. Since it is more convenient to work with 0-1 variables, we let $\chi(i) = (-1)^{X_i}$, where $X_i \in \{0,1\}$. Let

$$F(X) = -\sum_{A\in\mathcal{A}} \chi^k(A) = -\sum_{A\in\mathcal{A}} \sum_{i_1\in A} \cdots \sum_{i_k\in A} (-1)^{X_{i_1}+\cdots+X_{i_k}}.$$

Then finding a χ such that $\sum_{A\in\mathcal{A}} \chi^k(A)$ is at most its expectation is the same as finding an X such that $F(X) \geq E[F(X)]$.

2.2.6 Generating k-wise Independent Variables

It still remains to demonstrate a k-wise independent distribution on which we can perform a binary search efficiently in parallel.

Luby [49] gave the following such distribution for the case $k = 2$. Let $l = \lceil \log(n+1) \rceil + 1$ and $\omega = <\omega_1, \ldots, \omega_l>$ be picked uniformly from Z_2^l. Define random variables $X_1, \ldots, X_n$ such that

$$X_i = \left(\sum_{j=1}^{l-1} (i_j \omega_j) + \omega_l \right) \bmod 2,$$

where $<i_1, \ldots, i_{l-1}>$ is the binary expansion of i. Observe that Luby's distribution is not 4-wise independent: in particular, X_4, X_5, X_6, and X_7 are dependent since $X_7 = X_4 + X_5 + X_6$.

We extend Luby's distribution to be k-wise independent for all k as follows. We assign a label $a_i \in Z_2^l$ to each point i, where l is bounded by some polylogarithmic function of n. We pick $\omega \in Z_2^l$ uniformly at random, and let

$$X_i = a_i \cdot \omega.$$

Note that we can express Luby's distribution in this framework by letting $a_i = <i_1, \ldots, i_{l-1}, 1>$.

The main benefit of our distribution is that we can now give a necessary and sufficient condition for the X_i's to be independent and unbiased. (By unbiased, we mean each X_i has equal probability of being 0 or 1.) The following result is similar to others used in the literature [2]. For completeness, we provide a proof in what follows.

Theorem 2.2.13 *$X_{i_1}, \ldots, X_{i_k}$ are independent and unbiased if and only if $a_{i_1}, \ldots, a_{i_k}$ are linearly independent as vectors over Z_2.*

Proof First, we will prove the only if direction. Suppose we have k a's which are not linearly independent; i.e. we have $\sum_{j=1}^{k} \alpha_j a_{i_j} = 0$, where some $\alpha_{j^*} \neq 0$. Therefore,

$$\begin{aligned} \sum_{j=1}^{k} \alpha_j X_{i_j} &= \sum_{j=1}^{k} \alpha_j (a_{i_j} \cdot \omega) \\ &= (\sum_{j=1}^{k} \alpha_j a_{i_j}) \cdot \omega \\ &= 0 \cdot \omega \\ &= 0. \end{aligned}$$

But consider the event, E, that $X_{i_{j^*}} = 1$ and $X_{i_j} = 0$ for all $j \neq j^*$. Then

$$\sum_{j=1}^{k} \alpha_j X_{i_j} = \alpha_{j^*} \neq 0.$$

So E is not possible; i.e. $Pr[E] = 0$. But this implies that $X_{i_1}, \ldots, X_{i_k}$ are not independent, as any distribution where they were would have $Pr[E] = 2^{-k}$.

Now for the proof of the if direction. Assume $a_{i_1}, \ldots, a_{i_k}$ are linearly independent. Let A be the matrix containing $a_{i_1}, \ldots, a_{i_k}$ as row vectors. Then we have

$$A\omega = \begin{pmatrix} a_{i_1} \\ a_{i_2} \\ \vdots \\ a_{i_k} \end{pmatrix} \begin{bmatrix} \\ \omega \\ \\ \end{bmatrix} = \begin{bmatrix} X_{i_1} \\ X_{i_2} \\ \vdots \\ X_{i_k} \end{bmatrix}$$

Since $a_{i_1}, \ldots, a_{i_k}$ are linearly independent, we can create a new non-singular matrix $\hat{A}$ by augmenting A with $l-k$ new rows. Then $\hat{A} \cdot \omega = \hat{x}$, where the first k entries of $\hat{x}$ are $X_{i_1}, \ldots, X_{i_k}$. $\hat{A}$ is invertible, so $\omega = \hat{A}^{-1} \cdot \hat{x}$; hence, there is a 1-1 correspondence between $\hat{x}$'s and ω's implying that the $\hat{x}$'s are uniformly distributed. But there are exactly 2^{l-k} $\hat{x}$'s for every assignment to $X_{i_1}, \ldots, X_{i_k}$. So every assignment to $X_{i_1}, \ldots, X_{i_k}$ has probability $2^{l-k}/2^l = 2^{-k}$ of occurring. Therefore, $X_{i_1}, \ldots, X_{i_k}$ are independent and unbiased. □

To get $X_1, \ldots, X_n$ which are k-wise independent, we need a set of labels $a_1, \ldots, a_n$ such that every k of them are linearly independent. (By Theorem 2.2.13, this gives us k-wise independence of the X_i's.) In fact, it suffices to get an $n \times r$ matrix over $GF(2^s)$ with the property that any $\log n$ rows are linearly independent. Letting a_i be the ith row with each element $\alpha_0 + \alpha_1 x + \cdots + \alpha_{s-1} x^{s-1} \in GF(2^s)$ expanded out to $<\alpha_0, \ldots, \alpha_{s-1}>$ gives length $l = rs$ labels such that any $\log n$ are linearly independent over Z_2. Several well-known ways of getting such matrices, for $l = O(k \log n)$, are described in [2, 56]. Even randomly chosen labels of this length will work.

Theorem 2.2.14 *If $l > k \log(n/k) + 2k + t$, then a random set of labels $\{a_1, \ldots, a_n\} \subseteq Z_2^l$, are k-wise linearly independent with probability at least $1 - 2^{-t}$.*

Proof Pick $A = <a_1, \ldots, a_n> \in (Z_2^l)^n$ at random.

$$Pr[\text{every } k \text{ of } a_1, \ldots, a_n \text{ are linearly independent}]$$

$$\begin{aligned}
&\geq 1 - \sum_{\substack{S \subseteq \{1,\ldots,n\} \\ |S| = k}} Pr[\{a_i \mid i \in S\} \text{ is linearly dependent}] \\
&= 1 - \binom{n}{k} Pr[\exists \alpha_1, \ldots, \alpha_k \text{ not all 0 s.t. } \sum_{i=1}^{k} \alpha_i a_i = 0] \\
&\geq 1 - \binom{n}{k} \sum_{\substack{\alpha_1,\ldots,\alpha_k \in Z_2 \\ \text{not all 0}}} Pr[\sum_{i=1}^{k} \alpha_i a_i = 0]
\end{aligned}$$

Consider the a_i corresponding to the last non-zero α_i. Only one value for this a_i gives a 0 sum, so $Pr[\sum_{i=1}^{k} \alpha_i a_i = 0] = 2^{-l}$. Therefore, continuing the above sequence of relations, we have that

$$\begin{aligned}
&1 - \binom{n}{k} \sum_{\substack{\alpha_1,\ldots,\alpha_k \in Z_2 \\ \text{not all 0}}} Pr[\sum_{i=1}^{k} \alpha_i a_i = 0] \\
&= 1 - \binom{n}{k}(2^k - 1)2^{-l} \\
&\geq 1 - 2^{k(\log n + 1 - \log k) + k - 1} \\
&\geq 1 - 2^{-t}. \quad \square
\end{aligned}$$

Clearly, a random set of labels of length $l = k\log(4n)$ almost certainly gives us k-wise independence.

Since any k-wise independent distribution on n random variables must have a sample space of size $\Omega((n/k)^{\lfloor k/2 \rfloor})$ [2, 22], the labels $a_1, \ldots, a_n$ must be $\Omega(k\log n - k\log k)$ bits long.

2.2.7 Zeroing in on a Good Sample Point

Now that we have a k-wise independent distribution, we will explain how to do a binary search on it efficiently in parallel. We will employ the method of conditional probabilities, described earlier, for zeroing in on a "good" sample point; i.e. an ω such that $F(X) \geq E[F(X)]$. Even though the method is inherently sequential, since ω is only $O(\log^2 n)$ bits long, our resulting algorithm will be in NC.

To zero in on a good ω, one bit of ω is determined at a time, thereby performing a binary search on the ω's. This is done as follows. At the beginning of iteration t, assume we have set $\omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}$. Then we compute $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 0]$ and $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 1]$. We then set ω_t to the $s_t \in \{0,1\}$ which maximizes $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = s_t]$. We will show how to compute these conditional expectations in Section 2.2.8.

Lemma 2.2.15 *After step t of the above procedure,* $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t] \geq E[F(X)]$.

Proof (by induction on $|s|$)
The case $|s| = 0$ is clearly true. Assume this lemma is true for $t-1$; i.e. we have $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}] \geq E[F(X)]$. Then

$$
\begin{aligned}
& E[F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t] \\
&= \max(E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 0],\ E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 1]) \\
&\geq (E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 0] + E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}, \omega_t = 1])/2 \\
&= E[F(X) \mid \omega_1 = s_1, \ldots, \omega_{t-1} = s_{t-1}] \\
&\geq E[F(X)] \quad \text{(by inductive hypothesis).} \ \square
\end{aligned}
$$

Corollary 2.2.16 *The output of the above procedure is an X such that $F(X) \geq E[F(X)]$.*

2.2.8 Computing Conditional Expectations

In general, computing conditional expectations is hard to do and separates when one can and cannot use the method of conditional probabilities to zero in on a a good sample point. Fortunately, in the case of discrepancy, we have devised a simple and efficient approach for computing conditional expectations. Recall that to solve discrepancy, we need to compute conditional expectations $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t]$ where $F(X)$ is of the special form

$$F(X) = -\sum_{A \in \mathcal{A}} \sum_{i_1 \in A} \cdots \sum_{i_k \in A} (-1)^{X_{i_1} + \cdots + X_{i_k}}.$$

Using linearity of expected value, we can break this up into components

$$h_{i_1 \ldots i_k}(s) \quad = \quad E[(-1)^{\sum_{j=1}^{k} X_{i_j}} \mid \omega_1 = s_1, \ldots, \omega_t = s_t]$$

$$= E[(-1)^{\sum_{j=1}^{k} a_{i_j} \cdot \omega} \mid \omega_1 = s_1, \ldots, \omega_t = s_t]$$
$$= E[(-1)^{\bar{a} \cdot \omega} \mid \omega_1 = s_1, \ldots, \omega_t = s_t]$$

where $\bar{a} = \sum_{j=1}^{k} a_{i_j}$. Let r be the last position which contains a 1 in $\bar{a}$. If $t < r$, then $\bar{a} \cdot \omega$ is unbiased, and therefore $h_{i_1 \ldots i_k}(s) = 0$. Otherwise, $\bar{a} \cdot \omega$ is the same for all ω which extend s, and hence $h_{i_1 \ldots i_k}(s) = (-1)^{\bar{a} \cdot \omega}$. Assuming we have precomputed $\bar{a}$ and r, we can compute $h_{i_1 \ldots i_k}(s)$ in constant time by extending a partial sum $\sum_{j=1}^{t} \bar{a}_j s_j$ at each iteration and outputting $h_{i_1 \ldots i_k}(s) = 0$ if $t < r$ and $h_{i_1 \ldots i_k}(s) = (-1)^{\sum_{j=1}^{t} \bar{a}_j s_j}$ if $t \geq r$.

To compute $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t]$, we need one processor for each possible combination $<A, i_1, \ldots, i_k>$, that is, at most $n\Delta^k$ total. Therefore, k must be $O(\frac{\log n}{\log \Delta})$. Letting k be the minimum possible, $2\lceil \frac{\log n}{2\epsilon \log \Delta} \rceil$, implies that $n^{3+1/\epsilon}$ processors is sufficient.

Then we can compute all $h_{i_1 \ldots i_k}(s)$ terms in parallel in constant time and sum them to get $E[F(X) \mid \omega_1 = s_1, \ldots, \omega_t = s_t]$ in $O(\log n)$ time. Thus we spend $O(l \log n)$ time in the l iterations of our procedure. In addition, we can perform the precomputation required above in $O(l \log n)$ time as well. Since $l = O(\log^2 n)$, this yields an $O(\log^3 n)$ algorithm for discrepancy.

2.2.9 Application to Edge Coloring

An *edge coloring* of a graph $G = (V, E)$ is an assignment of colors to all edges of the graph, so that any two edges that share a common vertex are assigned different colors. Let Δ be the maximum degree in G. Observe that any coloring requires at least Δ colors. In fact, Vizing [65] implicitly gives a polynomial time algorithm to $\Delta + 1$ color any graph. Karloff and Shmoys [41] provide a parallel implementation of this algorithm to get a $\Delta + 1$ coloring of any graph in time $O(\Delta^{O(1)} \log^{O(1)} n)$ using a polynomial number of processors. Also of interest, there exist *NC* algorithms for optimally coloring bipartite graphs with Δ colors [28, 45, 25, 4]. Furthermore, there is a trivial *NC* algorithm to $2\Delta - 1$ color any graph by $\Delta + 1$ vertex coloring [49] the line graph. Berger and Shor [16] and Karloff and Naor [40] found *NC* algorithms to $\Delta + \Delta/\log^{O(1)} n$ color any graph.

Of particular interest here, there is an *RNC* algorithm in [41] which $\Delta + \Delta^{1/2+\epsilon}$ colors any graph. We will remove the randomness from this algorithm by using the techniques discussed

above. The *RNC* algorithm, *Algorithm A*, is as follows:

1. If $\Delta < (\log n)^{1/\epsilon}$, then use the Karloff–Shmoys $\Delta + 1$ deterministic algorithm [41].
2. Run an *RNC* algorithm for graph discrepancy, randomly picking χ until $disc(\chi) \leq \Delta^{1/2+\epsilon}$. Let $A = \{v \mid \chi(v) = +1\}$ and $B = \{v \mid \chi(v) = -1\}$. This partition gives us two graphs, both with vertex set V: bipartite graph G_1, which has all the edges of G between A and B; graph G_2 which has all the other edges of G.
3. Color G_1 using the Δ coloring algorithm for bipartite graphs.
4. Run *Algorithm A* recursively on G_2, using a new set of colors.

This algorithm works because the above partition implies that both G_1 and G_2 have maximum degree at most $\Delta/2 + \Delta^{1/2+\epsilon}$.

To make *Algorithm A* deterministic, we need only demonstrate a deterministic method for graph discrepancy, which we said in Section 2.2.1 was a special case of the set discrepancy problem. Plugging in the set discrepancy results with $\epsilon' = \epsilon/2$, we get a χ such that $disc(\chi) \leq \Delta^{1/2+\epsilon'}\sqrt{\log n}$. Note that $\Delta \geq (\log n)^{1/\epsilon}$, since we handled the other case in Step 1 of *Algorithm A*. Thus, $\sqrt{\log n} \leq \Delta^{\epsilon/2}$. So we have $disc(\chi) \leq \Delta^{1/2+\epsilon}$, which implies $\chi(N(v)) \leq \Delta^{1/2+\epsilon}$ for all $v \in V$.

2.3 Setting up a General Framework

For discrepancy-based problems, we considered a very specific class of functions, namely those of the form $\sum_{i=1}^{n'}(-1)^{\sum_{j=1}^{k} X_{i,j}}$, and showed how to achieve the expected value for these. What can we do in general? In particular, for which functions can we compute conditional expectations (the method of Section 2.2.7 will then apply to achieve the expected value)? In order to give ourselves a fighting chance, we will restrict our attention to functions of the form

$$F(X) = \sum_{i=1}^{m} f_i(X_{i,1}, X_{i,2}, \ldots, X_{i,k}).$$

These are exactly the functions for which we can apply Lemma 2.2.3 to show that k-wise independence gives the same expected value as full independence. Since we require at least one processor for each f_i term, we insist that m be polynomial in n. In Section 2.3.1, we will show

how to compute conditional expectations for arbitrary f_i when $k = O(\log n)$. In Section 2.3.2, we will describe the f_i's for which we can handle the case $k = O(\log^c n)$.

2.3.1 Logarithmic Number of 0/1 Variables

In this section, we will present two different methods for computing conditional expectations for functions of the form

$$F(X) = \sum_{i=1}^{n^a} f_i(X_{i,1}, \ldots, X_{i,b\log n}).$$

We present both methods because, depending on what problem we wish to solve, either of the two methods may be more efficient.

The first method is to rewrite F to be of the form solved in Section 2.2.8. Let $g : 2^{\{1,\ldots,k\}} \to \mathbf{R}$ be such that

$$g(A) = f_i(X_{i,1}, \ldots, X_{i,k}) \text{ such that } X_{i,j} = \begin{cases} 1 & \text{if } j \in A \\ 0 & \text{if } j \notin A \end{cases}$$

(i.e. g of a set is f_i applied to its characteristic vector). The next proposition follows from the theory of harmonic analysis on the cube.

Proposition 2.3.1 *[39]* $g(A) = \sum_{S \subseteq \{1,\ldots,k\}} \alpha_S (-1)^{|S \cap A|}$, *where* $\alpha_S = 2^{-k} \sum_{B \subseteq \{1,\ldots,k\}} g(B)(-1)^{|S \cap B|}$.

Thus,

$$\begin{aligned} f_i(X_{i,1}, \ldots, X_{i,k}) &= g(\{j \mid X_{i,j} = 1\}) \\ &= \sum_S \alpha_S (-1)^{|\{j \mid X_{i,j}=1\} \cap S|} \quad \text{(by Proposition 2.3.1)} \\ &= \sum_S \alpha_S (-1)^{\sum_{j \in S} X_{i,j}}. \end{aligned}$$

Since we have now written F as $\sum_{i=1}^{n^{a+b}} \alpha_i (-1)^{\sum_j X_{i,j}}$, we can apply the technique of Section 2.2.8 to compute conditional expectations. This gives us the following theorem:

Theorem 2.3.2 *There is an* NC *algorithm which given any* $F : Z_2^n \to \mathbf{R}$ *of the form*

$$F(X) = \sum_{i=1}^{n^a} f_i(X_{i,1}, \ldots, X_{i,b\log n}),$$

outputs an X *with* $F(X) \geq E[F(X)]$.

An alternative method for computing conditional expectations for F is as follows. First note that, by linearity of expectation, it suffices to compute the conditional expectations of the individual f_i and sum. Assume we wish to compute $E[\, f_i(X_{i,1}, \ldots, X_{i,b\log n}) \mid \omega_1 = s_1, \ldots, \omega_t = s_t \,]$. Let x be the vector $<X_{i,1}, \ldots, X_{i,b\log n}>$, and let A be the matrix whose rows are the corresponding labels $a_{i,1}, \ldots, a_{i,b\log n}$. Then $x = A\omega$. So

$$E[\, f_i(X_{i,1}, \ldots, X_{i,b\log n}) \mid \omega_1 = s_1, \ldots, \omega_t = s_t \,] = \sum_x f_i(x) Pr[\, A\omega = x \mid \omega_1 = s_1, \ldots, \omega_t = s_t \,].$$

If we let $\omega' = <\omega_1, \ldots, \omega_t>$, $\omega'' = <\omega_{t+1}, \ldots, \omega_l>$, A' and A'' be the first t and last $l-t$ columns of A respectively, and $s = <s_1, \ldots, s_t>$, then

$$\begin{aligned} E[\, f_i(x) \mid \omega_1 = s_1, \ldots, \omega_t = s_t \,] &= \sum_x f_i(x) Pr[\, A'\omega' + A''\omega'' = x \mid \omega' = s \,] \\ &= \sum_x f_i(x) Pr[\, A''\omega'' = x - A's \,]. \end{aligned}$$

For each x, we can test if the linear system $A''\omega'' = x - A's$ is solvable; if it is $Pr[\, A''\omega'' = x - A's \,] = 2^{-\text{rank}(A'')}$, otherwise $Pr[\, A''\omega'' = x - A's \,] = 0$. Since we can compute the contribution of each of the x's in parallel, we can compute the desired conditional expectation in NC, thus giving an alternate proof for Theorem 2.3.2.

2.3.2 Polylogarithmic Number of 0/1 Variables

Now we will consider the case of functions depending on a polylogarithmic number of variables. A simple counting argument shows that in NC we cannot compute all functions of $\log^c n$ variables, when $c > 1$, let alone compute conditional expectations of them. In fact, both techniques of Section 2.3.1 require evaluating f_i at every point; if f_i depends on more than a logarithmic number of variables, there will be a superpolynomial number of points to evaluate. However, there are some special cases for which we can compute conditional expectations.

The first special case we can handle is

$$f_i(X_{i,1}, \ldots, X_{i,k}) = (-1)^{\sum_j X_{i,j}}.$$

This function can be evaluated using the techniques of Section 2.2.8, even if $k = \log^c n$. In Section 2.3.1, we showed how to transform any function into a linear combination of these; if

this transformation is already known and provides only a polynomial number of non-zero α's, we can use this technique.

The next special case is based on the second technique of Section 2.3.1. Recall, we had

$$E[\, f_i(X_{i,1}, \ldots, X_{i,k}) \mid \omega_1 = s_1, \ldots, \omega_t = s_t \,] = \sum_x f_i(x) Pr[\, A''\omega'' = x - A's \,].$$

We can restrict our attention to those x for which $f_i(x) \neq 0$. If there are a polynomial number of these, we can compute conditional expectations of f_i for $k = \log^c n$. Some examples of this are logical AND and NOR of a polylogarithmic number of variables (each has one non-zero point).

A variant of the above, $f_i(X_{i,1}, \ldots, X_{i,k}) = X_{i,1}X_{i,2}\cdots X_{i_k}$ (i.e. the special case of monomials), was subsequently considered by Motwani et al. [54]. This is equivalent to the logical AND just described. Note that handling monomials is strictly weaker than the case above since, for example, it is impossible to write a poly-log variable NOR as a linear combination of a polynomial number of monomials.

Finally, we give a type of f_i which can simulate all the above and more. Consider functions of the form

$$f_i(x) = \begin{cases} 1 & \text{if } x = y_i + T_i z \text{ for some } z \in Z_2^k \\ 0 & \text{otherwise} \end{cases}$$

for some $y_i \in Z_2^k$, $T_i \in Z_2^{k \times k}$.

These are the characteristic functions of affine subspaces. Included are characteristic functions of all single points; we can write any function with a polynomial number of non-zero points as a linear combination of these. Functions $(-1)^{\sum_j X_{i,j}}$ can also be put in this form. To compute conditional expectations, we use a variant of our linear algebra method:

$$\begin{aligned} E[\, f_i(x) \mid \omega_1 = s_1, \ldots, \omega_t = s_t \,] &= Pr[\, A\omega = y_i + T_i z \text{ has a solution} \mid \omega' = s \,] \\ &= Pr[\, A''\omega'' + T_i z = y_i'' - A's \text{ has a solution} \,], \end{aligned}$$

which can be computed by performing Gaussian Elimination to determine how many bits of ω'' are free to vary. This gives us the following theorem:

Theorem 2.3.3 *There is an* NC *algorithm which given any* $F : Z_2^n \to \mathbf{R}$ *of the form*

$$F(X) = \sum_{i=1}^{n^a} f_i(X_{i,1}, \ldots, X_{i,b\log^c n}),$$

where each f_i is the characteristic function of some affine subspace of $Z_2^{\log^c n}$, outputs an X with $F(X) \geq E[F(X)]$.

2.4 Handling Multivalues — the Hypergraph Coloring Problem

In the previous sections, we were only concerned with the case where the random variables took on values 0 and 1 each with probability 1/2. Yet, for many problems, this model is too restrictive. In this section, we expand our framework to consider random variables drawn from a uniform distribution over a larger set of values. This can then be used to simulate non-uniform distributions. We will demonstrate our techniques for handling multivalued random variables on the following problem:

A *hypergraph* $\mathcal{H} = (V, \mathcal{E})$ is a system $\mathcal{E}$ of subsets of V called *edges*. $\mathcal{H}$ is d-uniform if every edge has d elements. Erdös and Kleitman [27] and Alon, Babai, and Itai [2] define the *large d-partite subhypergraph* problem as follows. Given a d-uniform hypergraph $\mathcal{H} = (V, \mathcal{E})$, find a d-coloring of V such that the number of edges in $\mathcal{E}$ having precisely one vertex of each color is at least $|\mathcal{E}|d!/d^d$. Alon et al. [2] showed this problem is in NC for constant d. We will show in this section that this problem is in NC for all d. Since the case of $d > \ln|\mathcal{E}| + \Omega(\lg\lg|\mathcal{E}|)$ is trivially satisfied by any coloring that colors one hyperedge correctly, we will henceforth restrict our attention to the case $d \leq \ln|\mathcal{E}| + O(\lg\lg|\mathcal{E}|)$.

2.4.1 Randomized Algorithm

In this section, we give a randomized parallel algorithm and prove that the expected number of properly colored edges is as desired. The randomized algorithm is as follows. Randomly assign to each vertex an $l = \log(2|\mathcal{E}|d!d^2) = O(\log|\mathcal{E}|\log\log|\mathcal{E}|)$ bit label. Designate these as random variables $Y = Y_1, \ldots, Y_{|V|}$. Let $\rho = \left\lfloor 2^l/d \right\rfloor$. A vertex is mapped to color i if its label is in $C_i = \{(i-1)\rho, \ldots, i\rho - 1\}$. Note that every color has ρ values associated with it. Vertices with values in the range $d\rho$ to $2^l - 1$ are uncolored. Note that fewer than d of the 2^l possible values yield an uncolored node.

Now for the analysis. We define a benefit function, $G(Y)$, which is the sum of terms $g_e(Y)$,

one for each $e \in \mathcal{E}$. Each $g_e(Y)$ is 1 if the vertices of edge e are assigned d different colors, and 0 otherwise.

In calculating the expected value of $g_e(Y)$, we get

$$\begin{aligned} E[g_e(Y)] &\geq Pr[g_e(Y) = 1 \mid \text{all vertices on edge } e \text{ properly colored}] \\ &\quad \times Pr[\text{all vertices on edge } e \text{ properly colored}] \\ &\geq \frac{d!}{d^d}\left(1 - \frac{d^2}{2^l}\right) \end{aligned}$$

Therefore,

$$\begin{aligned} E[G(Y)] &= |\mathcal{E}|\frac{d!}{d^d} - \frac{|\mathcal{E}|d!d^2}{d^d 2^l} \\ &> \frac{|\mathcal{E}|d! - 1}{d^d} \quad (\text{since } l > \log(2|\mathcal{E}|d!d^2)). \end{aligned}$$

Then, since $G(Y)$ is integral, $G(Y) \geq E[G(Y)]$ implies that $G(Y) \geq |\mathcal{E}|d!/d^d$, which is exactly what is desired.

2.4.2 The Basic Approach

In this section, we discuss various approaches for determinizing algorithms which use multivalued random variables. These approaches have different advantages and disadvantages and may all prove useful in applications.

The easiest approach to handling functions of multivalued random variables is to represent each variable by a collection of boolean random variables. In particular, for the large d-partite subhypergraph problem, if d is a power of 2, $d = O(\log|\mathcal{E}|/\log\log|\mathcal{E}|)$, we can represent the color of each vertex by $\lg d$ boolean random variables. Then each g_e would become a function of $d \lg d = O(\log|\mathcal{E}|)$ boolean variables, allowing us to apply the general framework of Section 2.3 to find a good coloring.

A second approach we might consider would be to replace Z_2 in our distribution with some other finite field $GF(q)$. For the large d-partite subhypergraph problem, if d is any prime power, $d = O(\log|\mathcal{E}|/\log\log|\mathcal{E}|)$, we can replace Z_2 with $GF(d)$. Theorem 2.2.13 will still hold and all of the approaches to get labels can be easily modified to work, giving us a distribution with $(\log n)$-wise independent random variables uniformly distributed over $GF(d)$. Since d is small (we only require polynomial in n), we can try each possible value for the next element of ω

in parallel and pick the one with the best conditional expected benefit G. To evaluate the conditional expectations, we can still use the linear algebra method of Section 2.3.1 to find the probability a collection of d random variables take on some particular value. We can do this for each possible value, since d^d is polynomial in n. Thus we can still efficiently zero in on a good sample point.

To find a good coloring for any d up to $\log|\mathcal{E}|$, we must use a more complicated approach, one which is similar to the one used by Luby for $\Delta + 1$ vertex coloring [49]. In essence, we repeatedly use the 0/1 problem as a subroutine to set one bit of the random variables at a time. We have multivalued random variables $Y = Y_1, \ldots, Y_{|V|}$ where $Y_i = Y_{i1}Y_{i2}\cdots Y_{il}$. We compute the Y_i's bit by bit. At step t, we compute $X^{(t)}$ such that

$$E[G(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \le j \le t] \ge E[G(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \le j \le t-1].$$

If we let

$$F^{(t)}(X^{(t)}) = E[G(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \le j \le t],$$

then the above is equivalent to finding an $X^{(t)}$ with $F^{(t)}(X^{(t)}) \ge E[F^{(t)}(X^{(t)})]$. Letting

$$f_e^{(t)}(X^{(t)}) = E[g_e(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \le j \le t]$$

allows us to write $F^{(t)}(X^{(t)})$ as a sum of $|\mathcal{E}|$ functions, each depending on at most $d \le \lg|\mathcal{E}|$ random variables $X_i^{(t)}$. Assuming that, given $X^{(1)}, \ldots, X^{(t-1)}$, we can construct functions $f_e^{(t)}$ (we will show how to do this in the next section) we can find a good $X^{(t)}$ using the general framework of Section 2.3.1.

A simple inductive argument shows that for all t,

$$E[G(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \le j \le t] \ge E[G(Y)].$$

It follows that letting Y be such that $Y_{ij} = X_i^{(j)}$ for all i and j implies that $G(Y) \ge E[G(Y)]$.

2.4.3 The Deterministic Algorithm

To apply the last multivalued approach described in the previous section, we must show how to construct, for any t and for any settings of the first $t-1$ bits $X^{(1)}, \ldots, X^{(t-1)}$, functions

$$f_e^{(t)}(X^{(t)}) = E[g_e(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \le j \le t].$$

To do so, we show how to compute

$$E[g_e(Y)|Y_{ij} = X_i^{(j)} \text{ for } 1 \leq j \leq t];$$

it then suffices to plug in the given $X^{(1)}, \ldots, X^{(t-1)}$ and every possible setting of the variables $\{X_i^{(t)} | i \in e\}$ to construct $f_e^{(t)}$.

Given edge e and the first t bits of each label, $f_e^{(t)}$ is the probability that the edge is properly colored. To calculate $f_e^{(t)}$, we sort the vertices of edge e into groups having the same t-bit prefix. For each t bit string α, we let S_α be the set of vertices which have prefix α and let I_α be the set of 2^{l-t} values which have prefix α. We let T_α be $\{1 + \sum_{\beta<\alpha} |S_\beta|, \ldots, \sum_{\beta \leq \alpha} |S_\beta|\}$. Observe that edge e is properly colored if and only if for each α the vertices in S_α are assigned the colors in T_α.

Now we can calculate $f_e^{(t)}$ as follows:

$$\begin{aligned} f_e^{(t)}(X^{(t)}) &= \prod_{|\alpha|=t} Pr[\text{vertices in } S_\alpha \text{ are assigned colors in } T_\alpha] \\ &= \prod_{|\alpha|=t} |S_\alpha|! \prod_{i \in T_\alpha} Pr[\text{vertex in } S_\alpha \text{ gets color } i] \\ &= \prod_{|\alpha|=t} |S_\alpha|! \prod_{i \in T_\alpha} |C_i \cap I_\alpha| / 2^{l-t}. \end{aligned}$$

Theorem 2.4.1 *The large d-partite subhypergraph problem, finding a coloring of V which properly colors at least $|\mathcal{E}|d!/d^d$ edges, is in* NC.

2.5 Improved Discrepancy Algorithms and Bounds

The techniques and results for discrepancy-related problems can be improved in several ways. For example, in this section we show how, in *NC*, to bound the discrepancy of each set in terms of its size (rather than the size of the largest set), solve weighted discrepancy, and match the sequential discrepancy bound when Δ is polylogarithmic. These improvements have additional applications such as lattice approximation.

2.5.1 Improved Discrepancy Bound for Variable-sized Sets

In Section 2.2, we bounded discrepancy in terms of Δ, the maximum cardinality of any set $A \in \mathcal{A}$. Here, on the other hand, we will achieve, for each $A \in \mathcal{A}$, a similar bound on $\chi(A)$ in terms

of $|A|$. To do so, it is more convenient to reformulate the discrepancy problem as follows: given vectors $v_1, \ldots, v_n \in \{0,1\}^n$, find an $x \in \{-1,+1\}^n$ such that for all i, $|v_i \cdot x| \leq \Delta^{1/2+\epsilon}\sqrt{\log n}$, where $\Delta = \max_i ||v_i||_1$. As this is equivalent to set discrepancy, the algorithm presented in Section 2.2 can be easily recast to solve this problem. The existing algorithm will in fact work even if we allow the v_i's to be in $\{-1,0,+1\}^n$ — the key observation being that Lemma 2.2.11 will still hold.

Whereas the discrepancy bound in Section 2.2 is achieved in terms of the maximum L_1-norm, here we will achieve the bound in terms of the L_1-norm of each vector.

Theorem 2.5.1 *Given vectors $v_1, \ldots, v_n \in \{-1,0,+1\}^n$, there exists an* NC *algorithm to find an $x \in \{-1,+1\}^n$ such that for all i, $|v_i \cdot x| \leq ||v_i||_1^{1/2+\epsilon}\sqrt{\log n}$.*

Proof We modify our discrepancy algorithm as follows: we let

$$F(X) = \sum_{i=1}^{n} \frac{(v_i \cdot x)^{k_i}}{E[(v_i \cdot x)^{k_i}]},$$

where $k_i = 2\lceil \frac{\log n}{2\epsilon \log ||v_i||_1} \rceil$. Then getting $F(X) \leq E[F(X)] = n$ implies $(v_i \cdot x)^{k_i} \leq n\, E[(v_i \cdot x)^{k_i}] \leq n(k_i||v_i||_1)^{k_i/2}$. This, in turn, implies $|v_i \cdot x| \leq ||v_i||_1^{1/2+\epsilon}\sqrt{\log n}$. □

2.5.2 Weighted Discrepancy

Still adhering to the notation introduced in the previous section, we will now consider the *weighted discrepancy* problem, i.e. the case when the entries of the v_i's are arbitrary real weights between -1 and $+1$. Spencer [60, 61] provides a poly-time algorithm for this problem which has the same performance as the poly-time unweighted case. To obtain an *NC* algorithm, we will reduce weighted discrepancy to the unweighted case we considered above.

Theorem 2.5.2 *Given vectors $v_1, \ldots, v_n \in \mathbf{R}^n$, $||v_i||_\infty \leq 1$, there exists an* NC *algorithm to find an $x \in \{-1,+1\}^n$ such that for all i, $|v_i \cdot x| \leq O(||v_i||_1^{1/2+\epsilon}\sqrt{\log n})$.*

Proof Without loss of generality, we can assume the v_i's are normalized so that $||v_i||_\infty = 1$ for all i. We can then round each entry to $\log n$ bits. This will induce a total error of at most 1, which is negligible since $||v_i||_1 \geq 1$. Next, by taking the binary expansions, we replace each v_i with $\log n$ vectors $v_{i0}, \ldots, v_{i,\log n} \in \{-1,0,+1\}^n$ such that $v_i = \sum_{j=0}^{\log n} 2^{-j} v_{ij}$ and $||v_i||_1 =$

$\sum_{j=0}^{\log n} 2^{-j}||v_{ij}||_1$. Finally, we apply the discrepancy algorithm of Theorem 2.5.1 to the v_{ij}'s (with the $n \log n$ vectors padded out with zeroes to length $n \log n$). The x returned is such that for all i

$$\begin{aligned}
|v_i \cdot x| &= |\sum_j 2^{-j}(v_{ij} \cdot x)| \\
&\leq \sum_j 2^{-j}|v_{ij} \cdot x| \\
&\leq \sum_j 2^{-j}||v_{ij}||_1^{1/2+\epsilon}\sqrt{\log(n \log n)} \\
&= O(\sqrt{\log n}\sum_j (2^{1/2-\epsilon})^{-j}(2^{-j}||v_{ij}||_1)^{1/2+\epsilon}) \\
&\leq O(\sqrt{\log n}\sum_j (2^{1/2-\epsilon})^{-j}||v_i||_1^{1/2+\epsilon}) \\
&= O(||v_i||_1^{1/2+\epsilon}\sqrt{\log n}). \quad \square
\end{aligned}$$

An application of weighted discrepancy is *lattice approximation*: given vectors $v_1, \ldots, v_n \in [-1,1]^n$ and $x \in [0,1]^n$, find an $\hat{x} \in \{0,1\}^n$ such that for all i, $|v_i \cdot (x - \hat{x})|$ is small. Several researchers provide poly-time algorithms for this problem. Beck and Fiala [7] and Raghavan [57] use the method of conditional probabilities to construct algorithms that find an $\hat{x}$ such that every $|v_i \cdot (x - \hat{x})|$ is bounded by $O(\sqrt{n \log n})$ and $O(\sqrt{||v_i||_1 \log n})$ respectively. Beck and Spencer [8] (see also [61, p. 40–42]) show how to reduce the lattice approximation problem to the discrepancy problem, obtaining an algorithm which outputs an $\hat{x}$ such that, for all i, $|v_i \cdot (x - \hat{x})| \leq O(\sqrt{i} \log i)$. Motwani et al. [54] apply the Beck-Spencer reduction to get an *NC* algorithm for the special case where $v_1, \ldots, v_n \in \{0,1\}^n$; their algorithm outputs an $\hat{x}$ such that $|v_i \cdot (x - \hat{x})| \leq O(\Delta^{1/2+\epsilon}\sqrt{\log n})$, where $\Delta = \max_{i=1}^n ||v_i||_1$. Using the algorithm of Theorem 2.5.2 for the more general case of weighted discrepancy, the Beck-Spencer reduction can be immediately applied to obtain *NC* lattice approximation in its most general form.

Corollary 2.5.3 *Given vectors $v_1, \ldots, v_n \in [-1,1]^n$ and $x \in [0,1]^n$, there exists an* NC *algorithm to find an $\hat{x} \in \{0,1\}^n$ such that for all i, $|v_i \cdot (x - \hat{x})| \leq O(||v_i||_1^{1/2+\epsilon}\sqrt{\log n})$.* $\square$

2.5.3 An $O(\sqrt{\Delta \log n})$ Discrepancy Algorithm for Small Δ

The discrepancy algorithm of Section 2.2 can be improved to yield a $2\sqrt{\Delta \log n}$ bound in the special case where $\Delta = \log^c n$. The improved algorithm, besides achieving a better discrepancy

bound, is a nice example of how to apply the techniques of this chapter.

Theorem 2.5.4 *There exists a parallel algorithm using $O(\Delta^2 \log^3 n)$ time on $O(n^2 \log^c n)$ processors, which, given a set system $\mathcal{A}$ with maximum degree Δ, outputs a χ with $disc(\chi) \leq 2\sqrt{\Delta \ln n}$.*

Proof For each $A \in \mathcal{A}$, we let $g_A(X)$ be 0 if $|\chi(A)| \leq 2\sqrt{\Delta \ln n}$, and 1 otherwise. Let $G(X) = \sum_{A \in \mathcal{A}} g_A(X)$, i.e. $G(X)$ is the number of unbalanced sets. We want to find an X such that $G(X) \leq E[G(X)] < 1$. To do this we use an approach similar to the one in Section 2.4 for multivalues. We first partition Γ into $r = O(\Delta^2)$ subsets $\Gamma_1, \ldots, \Gamma_r$ such that the intersection of each Γ_j with any $A \in \mathcal{A}$ is less than $\log n / \log \Delta$ (we will show how to do this below). Then for each j in sequence, we can construct a function $F^{(j)}(X^{(j)})$, where $X^{(j)}$ are the random variables corresponding to Γ_j, which is the expected value of $G(X)$ conditioned upon the values of $X^{(1)}, \ldots, X^{(j-1)}$ set already and the given $X^{(j)}$. Each $F^{(j)}$ is a sum of functions depending on at most $\log n / \log \Delta$ variables each, so we can apply our general framework to find a good $X^{(j)}$. A simple inductive argument shows that when we are done, we have a good X.

It remains to show how to construct $\Gamma_1, \ldots, \Gamma_r$ such that the intersection of each Γ_j with any $A \in \mathcal{A}$ is less than $\log n / \log \Delta$. We think of coloring the elements of Γ with $O(\Delta^2)$ colors such that no $A \in \mathcal{A}$ has $\log n / \log \Delta$ or more elements of any one color. Equivalently, we want to color so that no $(\frac{\log n}{\log \Delta})$-subset of any $A \in \mathcal{A}$ is monochromatic.

To accomplish this, we let $Y =< Y_1, \ldots, Y_n >$, where Y_i is a random variable taking on values $1, \ldots, \Delta^2$ uniformly. Let

$$H(Y) = \sum_{A \in \mathcal{A}} \sum_{\substack{B \subseteq A \\ |B| = \log n / \log \Delta}} h_B(Y),$$

where $h_B(Y)$ is 1 if B is monochromatic, and 0 otherwise. Note that $H(Y)$ represents the number of monochromatic $(\frac{\log n}{\log \Delta})$-subsets, when coloring Γ according to Y.

We will first show how to get a setting of Y such that at most Δ^2 of the subsets are monochromatic, and then we will eliminate this entirely. We begin by computing $E[H(Y)]$. We

note that, by linearity of expected value,

$$\begin{aligned} E[H(Y)] &= \sum_{A \in \mathcal{A}} \sum_{\substack{B \subseteq A \\ |B| = \log n / \log \Delta}} E[h_B(Y)] \\ &\leq n \binom{\Delta}{\frac{\log n}{\log \Delta}} (\Delta^2)^{1 - \log n / \log \Delta} \\ &\leq n^2 \frac{\Delta^2}{n^2} \\ &= \Delta^2. \end{aligned}$$

Note that $H(Y)$ is a sum of terms depending on $\log n / \log \Delta$ Y_i's each, and that each Y_i can be represented as $2 \log \Delta$ binary random variables. Thus, we can apply our general framework to find a Y with $H(Y) \leq \Delta^2$. Alternatively, we can set the Y_i's one bit at a time calling the general framework as a subroutine, as in Section 2.4.2. The latter approach is more processor efficient.

Now we have a coloring such that at most Δ^2 of the subsets are monochromatic. We will take one element from each of the monochromatic $(\frac{\log n}{\log \Delta})$-subsets and give each a new color. This adds at most Δ^2 additional colors, and leaves no monochromatic $(\frac{\log n}{\log \Delta})$-subsets.

This discrepancy algorithm makes $\Delta^2 + 2 \log \Delta$ calls to our general framework (assuming one call for each bit of the Y_i's). Thus, the running time is $O(\Delta^2 \log^3 n)$. The number of processors is dominated by the partitioning phase, which can be done using $O(n^2 \log^c n)$ processors. □

Corollary 2.5.5 *There exists an* NC *algorithm which, given a set system $\mathcal{A}$ with maximum degree $\Delta \leq \log^c n$, outputs a χ with $disc(\chi) \leq 2\sqrt{\Delta \ln n}$.*

Chapter 3

Randomness in Interactive Proofs

3.1 Introduction

In this chapter, we consider the problem of sampling an arbitrary function. More specifically, we consider randomized polynomial-time procedures which output samples $x_1, \ldots, x_m \in \{0,1\}^l$, with the property that for any function f, with high probability, the average of f applied to the x_i's is very close to the average value of f over all inputs. Our goal is to implement such a procedure using as few random bits as possible.

Our solution involves a novel use of a property of low-independence distributions; namely, that it is inexpensive to produce a large (but somewhat "low quality") random sample. We will use an iterated sampling procedure which exploits tradeoffs between the size of the space sampled, the size of a sample, its independence, the quality of the sample, and the number of random bits needed. Instead of taking a random sample of m x_i's directly, we use a series of samples. Our first sample will be very large, but coarse (i.e. it will be based on very low independence). Then we take a smaller sample of this first sample. This second sample will be about the square root of the size of the first sample, but will be less coarse, using twice as much independence. We repeat, taking smaller but finer samples until we are left with just the desired m points.

We begin by discussing this sampling primitive and related work. We then describe the application which motivated its construction: randomness-efficient error-reduction for Arthur-

This chapter describes joint work with Mihir Bellare [12].

Merlin games.

3.1.1 Universal (l, ϵ, δ)-Sampling and Our Result

A universal (l, ϵ, δ)-sampler may be informally described as a randomized, polynomial-time process which outputs a sequence of m *sample points* $x_1, \ldots, x_m \in \{0,1\}^l$ such that: for any function $f : \{0,1\}^l \to [0,1]$ it is the case that

$$\Pr\left[\left|\frac{1}{m}\sum_{i=1}^{m} f(x_i) - E[f]\right| < \epsilon\right] \geq 1 - \delta\,.$$

Notice that we place no restriction on the function f (we do not require f to be polynomial-time computable, for example). Rather, we require that the sampler be able to approximate *any* function with high probability. The sampler is, in fact, independent of the function f, f is not even an input to the sampler, and in particular it does not even evaluate the function. This will be important to our application.

The straightforward procedure for universal sampling is of course to just select $m = O(\epsilon^{-2}\log\delta^{-1})$ independent and uniformly distributed sample points. This yields a universal (l, ϵ, δ)-sampler at the cost of $O(ml)$ coin tosses.

We will be interested in designing universal samplers which use few random bits. (A related concern is the number m of sample points which must remain polynomially bounded). Previous randomness-efficient universal sampling techniques have suffered from one of two drawbacks: either (1) they serve to approximate only a restricted class of functions (such as boolean functions [1],[24],[37]), or (2) they generate a number of sample points proportional to δ^{-1} (rather than $\log\delta^{-1}$) and thus cannot be used when the desired error probability is exponentially small. We present a construction which suffers from neither of these drawbacks: our main result is the construction of a universal (l, ϵ, δ)-sampler which outputs $poly(\epsilon^{-1}, \log\delta^{-1}, l)$ sample points using $O(l + \log\delta^{-1} \cdot \log l)$ coin tosses.

3.1.2 Previous and Related Work

In [21], Chor and Goldreich showed that dramatic savings in the number of coin tosses needed to construct a variant of the universal (l, ϵ, δ)-sampler can be achieved by selecting pairwise independent sample points. This extends to a construction for a universal (l, ϵ, δ)-sampler at

the cost of $2l$ coin tosses, by selecting $O(\epsilon^{-2}\delta^{-1})$ pairwise independent sample points. However, the number of sample points here grows inversely proportional to the desired error probability δ, and thus this method cannot be applied in polynomial time when the desired error probability is exponentially small.

An alternative universal sampling method for the special case of boolean valued functions (i.e. f takes on only the values 0 and 1) is based on selecting a random walk on a 2^l node explicitly constructed expander graph (cf. [1],[24],[37]). This method yields a universal $(l, \frac{1}{3}, \delta)$ sampler of boolean functions which outputs $O(\log \delta^{-1})$ sample points using $l + O(\log \delta^{-1})$ coin tosses[1].

A sampling primitive of a slightly different flavor was recently constructed by Goldreich [29]. He outputs a collection of $mz = O(\epsilon^{-2} \log \delta^{-1})$ sample points $x_1^1, \ldots, x_m^1, \cdots, x_1^z, \ldots, x_m^z$ grouped into $z = O(\log \delta^{-1})$ blocks of $m = O(\epsilon^{-2})$ points each with the property that, with probability $\geq 1 - \delta$, the average value of f on block j (i.e. $\sum_{i=1}^m f(x_i^j)$) differs from $E[f]$ by at most ϵ for a majority of the blocks j. His method uses only $O(l + \log \delta^{-1})$ coin tosses.

Although weaker than universal (l, ϵ, δ)-sampling, Goldreich's primitive can be used to implement the application to the randomness-efficient error reduction of Arthur-Merlin games which we discuss in Section 3.3.

3.2 Universal Sampling Using k-wise Independence

In this section we describe how to implement the universal sampling primitive using few random bits. We begin with a more precise specification of the primitive. Next we introduce the two major tools we will use: k-universal hash functions and the k-wise independence tail inequality.

We first construct, as an illustration of our methods, a simple universal sampler which nonetheless gives a non-trivial savings in coin tosses. We then present an iterated sampling technique which significantly reduces the number of random bits used. Finally we specify the sampler that results.

[1] One can obtain a universal (l, ϵ, δ)-sampler of arbitrary functions by using the ideas of [37], but this will require $\Omega(\epsilon^{-2} \log \delta^{-1})$ sample points generated using $l + \Omega(\epsilon^{-2} \log \delta^{-1})$ coin tosses.

3.2.1 Universal (l, ϵ, δ)-Sampling

Definition 3.2.1 Let $l : \mathbf{N} \to \mathbf{N}$ and $\epsilon, \delta : \mathbf{N} \to [0,1]$. A *universal* (l, ϵ, δ)*-sampler* is a randomized, polynomial time algorithm which on input 1^n outputs a sequence of points $x_1, \ldots, x_m \in \{0,1\}^{l(n)}$ such that: for any collection of m functions $f_1, \ldots, f_m : \{0,1\}^{l(n)} \to [0,1]$ it is the case that

$$\Pr\left[\left|\frac{1}{m}\sum_{i=1}^{m} f_i(x_i) - E[f_i]\right| < \epsilon(n)\right] \geq 1 - \delta(n)$$

(where $E[f_i] = 2^{-l(n)} \sum_{x \in \{0,1\}^{l(n)}} f_i(x)$).

Notice that this definition is slightly more general than what we discussed in Section 3.1.1 since we are talking about approximating a *collection* of functions rather than a single function. This will be important for our application.

The points $x_1, \ldots, x_m$ are called the *sample points*, and we refer to the sequence of coin tosses used by the sampler as the *seed*.

3.2.2 Universal Hash Functions

A k-universal family of hash functions [19] from m-bit strings to l-bit strings is a collection $\{h_i\}$ of polynomial-time computable functions from $\{0,1\}^m$ to $\{0,1\}^l$ such that

1. $\Pr[h_i(x_1) = y_1 \wedge \cdots \wedge h_i(x_k) = y_k] = 2^{-kl}$ for any distinct $x_1, \ldots, x_k \in \{0,1\}^m$ and any $y_1, \ldots, y_k \in \{0,1\}^l$ (i.e. $\{h(x)\}_{x \in \{0,1\}^m}$ are k-wise independent and uniformly distributed over $\{0,1\}^l$).

2. Given $j \leq k$ and any $x_1, \ldots, x_j \in \{0,1\}^m$ and $y_1, \ldots, y_j \in \{0,1\}^l$, it is possible in probabilistic polynomial-time to uniformly sample from the h_i such that $h_i(x_1) = y_1 \wedge \cdots \wedge h_i(x_j) = y_j$.

The standard construction of a k-universal family of hash functions from m-bit strings to m-bit strings is as follows. Let $F = \mathrm{GF}(2^m)$. For $a_0, \ldots, a_{k-1} \in F$, we let

$$h_{a_0,\ldots,a_{k-1}}(x) = a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}.$$

The fact that this is a k-universal hash function follows from a simple interpolation argument. Finally note that the case $m < l$ can be handled by padding out the argument to the hash function; the case $m > l$ can be handled by stripping bits from the result of the hash function.

We will use $H^k_{m,l}$ to denote the standard family of k-universal hash functions from m bits to l bits. Note that the number of bits required to encode an element of $H^k_{m,l}$ is $k \cdot \max(m, l)$.

3.2.3 The k-wise Independence Tail Inequality

In this chapter we will make heavy usage of the simple form of Lemma 2.2.7 described in Section 2.2.4, namely that if $k \geq 4$ is an even integer, $\{X_i\}_{i=1}^n$ is a collection of k-wise independent random variables in the range $[0, 1]$, $X = \sum_{i=1}^n X_i$, $\mu = E[X]$, and $A > 0$, then

$$\Pr[|X - \mu| > A] < \left(\frac{nk}{A^2}\right)^{k/2}.$$

We will refer to this as *the k-wise Independence Tail Inequality.*

3.2.4 A Simple Universal Sampler

Using k-wise independent hash functions we can construct a very simple universal (l, ϵ, δ)-sampler as follows. The sampler takes as input a randomly selected element h from $H^k_{d,l}$, where $d \geq \lg m$, and outputs $h(1), \ldots, h(m)$ (identifying $\{0,1\}^d$ with $\{1, 2, \ldots, 2^d\}$). We use the k-wise independence tail inequality to specify m and k.

Let $Y_i = f_i(h(i))$ for $1 \leq i \leq m$. It follows from the definition of k-wise independent hash functions that $\{Y_i\}_{i=1}^m$ is a collection of k-wise independent random variables in the range $[0, 1]$. Thus the k-wise independence tail inequality will hold to bound $Y = \sum_{i=1}^m Y_i$. In particular, assume k is an even integer ≥ 4. Then we have

$$\Pr[\,|Y - E[Y]| \geq \epsilon m\,] \leq \left(\frac{mk}{(\epsilon m)^2}\right)^{k/2} = \left(\frac{k}{\epsilon^2 m}\right)^{k/2}.$$

But the left hand side is just

$$\Pr\left[\,\left|\sum_{i=1}^m f_i(x_i) - \sum_{i=1}^m E[f_i]\right| \geq \epsilon m\,\right] = \Pr\left[\,\left|\frac{1}{m}\sum_{i=1}^m f_i(x_i) - E[f_i]\right| \geq \epsilon\,\right].$$

Since we want this probability to be less than δ, it suffices to have

$$\delta \geq \left(\frac{k}{\epsilon^2 m}\right)^{k/2}$$

or equivalently,

$$m \geq \frac{k}{\epsilon^2 \delta^{2/k}}.$$

Restating the above, we have the following lemma.

Lemma 3.2.2 *Let k, m, d be integers such that $k \geq 4$ is even, $m \geq \frac{k}{\epsilon^2\delta^{2/k}}$, and $d \geq \lg m$. Then for any collection of m functions $f_1, \ldots, f_m : \{0,1\}^l \to [0,1]$, picking h at random from $H^k_{d,l}$ implies that*

$$\Pr\left[\left|\frac{1}{m}\sum_{i=1}^{m} f_i(h(i)) - E[f_i]\right| < \epsilon\right] \geq 1 - \delta\,.$$

Next, observe that this sampler uses $k \cdot \max(d, l)$ bits. This leads us to think that we just make k as small as possible and m as large as necessary to minimize the number of bits. However, we have another constraint: m must be bounded by a polynomial in the input size. Requiring that m be polynomial in n and optimizing, we get $k = \frac{\log \delta^{-1}}{\log n^{O(1)}}$ and thus use $O\left(\frac{l \log \delta^{-1}}{\log n} + \log \delta^{-1}\right)$ random bits.

3.2.5 Iterated Sampling

In the previous section, we showed how to sample a collection of functions using k-wise independent hash functions. The number of bits we used to (l, ϵ, δ)-sample, for fixed ϵ and δ, was proportional to the logarithm of the domain size of the functions and roughly inversely proportional to the logarithm of the size of the sample. Given a set of functions with a fixed domain size, this suggested that we simply make our sample as large as is tolerable (i.e. polynomial).

In this section we will improve our bounds by iterating the universal sampling primitive of the previous section in a novel manner. Roughly, the idea is to take a large sample and then take a smaller sample of the first sample. Each of these samples will require many fewer bits than our original method: the first because the sample is larger; the second because we are sampling a smaller space. Our sampler becomes the composition of two randomly chosen hash functions. This idea of taking multiple samples can be improved by taking a sequence of smaller and smaller samples instead of just two.

It is important to note that we never actually compute all of the points of the samples other than the last one—they are simply defined as the ranges of hash functions. This means that these samples can be superpolynomial in size. Only the final sample is required to be polynomial sized.

Except for the final sample, we will think of sampling just a single function, instead of the m functions used in Lemma 3.2.2. We will therefore need the following variant of Lemma 3.2.2.

Lemma 3.2.3 *Let k, d be integers such that $k \geq 4$ is even and $2^d \geq \frac{k}{\epsilon^2\delta^{2/k}}$. Then for any function $f : \{0,1\}^l \to [0,1]$, picking h at random from $H^k_{d,l}$ implies that*

$$\Pr[\,|E[f \circ h] - E[f]| < \epsilon\,] \geq 1 - \delta\,.$$

Proof Let $m = 2^d$, and let $f_1 = f_2 = \cdots = f_m = f$. Then applying Lemma 3.2.2 gives the desired result. □

Now we can combine Lemmas 3.2.2 and 3.2.3 to obtain a new sampling lemma. Our sampler will be the composition of a sequence of length doubling hash functions. This represents a sequence of samples in which the size of each sample is the square root of the size of the preceding sample.

Lemma 3.2.4 *Let r, m, d be integers and $k_1, \ldots, k_r$ even integers ≥ 4. Suppose $d \geq \lg m$ and*

$$2^{2^{r-j}d} \geq \frac{k_j}{\epsilon^2\delta^{2/k_j}} \quad (j = 1, \ldots, r-1) \quad \text{and} \quad m \geq \frac{k_r}{\epsilon^2\delta^{2/k_r}}\,.$$

Then for any collection of m functions $f_1, \ldots, f_m : \{0,1\}^{2^r d} \to [0,1]$, picking h_j at random from $H^{k_j}_{2^{r-j}d, 2^{r-j+1}d}$ implies that

$$\Pr\left[\,\left|\tfrac{1}{m}\textstyle\sum_{i=1}^m (f_i \circ h_1 \circ \cdots \circ h_r)(i) - E[f_i]\right| < r\epsilon\,\right] \geq 1 - r\delta\,.$$

Proof Let $f = \frac{1}{m}\sum_{i=1}^m f_i$. We first claim by induction that for $0 \leq j \leq r-1$

$$\Pr[\,|E[(f \circ h_1 \circ \cdots \circ h_j)] - E[f]| < j\epsilon\,] \geq 1 - j\delta\,.$$

The base case ($j = 0$) is immediate, and the induction step is just Lemma 3.2.3 (using $f \circ h_1 \circ \cdots \circ h_{j-1}$ as the function). The final step is to apply Lemma 3.2.2 (using $\{f_i \circ h_1 \circ \cdots \circ h_{r-1}\}_{i=1}^m$ as the collection of functions). □

3.2.6 Our Universal Sampler

We now optimize the parameters of Lemma 3.2.4 to get a particular universal sampler. In this optimization, there is a trade-off between the number of sample points the sampler outputs and the number of random bits it uses to do this. It will be convenient to write our error in the form $\delta_1\delta_2$ where we assume δ_1 is $\leq n^{O(1)}$. This is convenient because we can allow the number of random bits to grow proportional to $\log \delta_2^{-1}$ by letting the number of sample points grow proportional to $\delta_1^{-1} \log \delta_2^{-1}$.

Theorem 3.2.5 *Suppose* $l : \mathbf{N} \to \mathbf{N}$ *is* $\leq n^{O(1)}$ *with* $\log n = o(l)$, *and* $\epsilon, \delta_1, \delta_2 : \mathbf{N} \to [0,1]$ *are* > 0 *with* $\epsilon^{-1}, \delta_1^{-1}, \log \delta_2^{-1} \leq n^{O(1)}$. *Then we can construct a universal* $(l, \epsilon, \delta_1\delta_2)$*-sampler which outputs* $m = O(\epsilon^{-6} \log^6 l + \delta_1^{-1} \log l + \log^3 \delta_2^{-1})$ *sample points using* $O(l + \log \delta_2^{-1} \cdot \log l)$ *coin tosses.*

Proof We apply Lemma 3.2.4. Let $m = \max(\epsilon^{-6} \log^6 l, \delta_1^{-1} \log l, [12(1 + \log \delta_2^{-1})]^3)$ and $r = \log(l/\log m)$. Let the ϵ of Lemma 3.2.4 be ϵ/r and the δ be $\delta_1\delta_2/r$, and let

$$k_j = \frac{12}{2^{r-j+1} \log m} \left(2^{r-j+1} \log m + \log \delta_2^{-1}\right) \quad (j = 1, \ldots, r).$$

The conditions of Lemma 3.2.4 can now be verified. □

3.3 Randomness-Efficient Error-Reduction for Arthur-Merlin Games

In this section, we apply the results of Section 3.2 to derive a randomness-efficient method of reducing the error probability of Arthur-Merlin proof systems. We begin with a review of Arthur-Merlin games and proof systems and the standard method of error-reduction. We then discuss the ideas of our protocol and the particular values of l, ϵ, and δ it requires, and conclude with a proof of our randomness-efficient error-reduction theorem. This section extends techniques first appearing in [10].

3.3.1 Arthur-Merlin Games

An *Arthur-Merlin game* is a two-party protocol played by an all-powerful "prover", called *Merlin*, and a polynomial-time "verifier", called *Arthur*. The game is played on a common input, and its purpose is to convince Arthur that the input belongs to some predetermined language. Arthur's role in the process is restricted to tossing coins, sending their outcome to Merlin, and finally evaluating a polynomial-time predicate applied to the common input and the full transcript of the interaction. Arthur-Merlin games are a special form of interactive proof systems [31], but their language recognition power has been shown to be equal to that of interactive proof systems [33].

Let w denote the common input to the (Arthur-Merlin) game, $n = |w|$ its length, $l(n)$ the length of Arthur's messages, $q(n)$ the length of Merlin's messages, and $g(n)$ the number

of rounds. We denote by $\rho(w, C) \in \{0, 1\}$ Arthur's *decision* on input w and conversation C. The conversation C can be parsed uniquely into Arthur's and Merlin's messages: $C = r^1 y^1 \cdots r^g y^g$, where r^t is Arthur's t-th message and y^t is Merlin's response (we assume without loss of generality that Arthur plays first and Merlin second in each round). A strategy for Arthur, $A = (\rho, g, l, q)$, consists of the decision predicate ρ, as well as (polynomially bounded) functions specifying the number of rounds and the length of messages sent in each round by each party. For sake of simplicity we assume that the length of the messages sent in each round is independent of the round.

Let M be a strategy for Merlin (i.e. M determines the next message of Merlin based on the common input and the messages received so far from Arthur). We denote by $\Pr[\,(A, M) \text{ accepts } w\,]$ the probability that $\rho(w, C) = 1$ when C is chosen at random (the probability space is that of all possible choices of $r^1, \ldots, r^{g(|w|)}$ taken with uniform distribution, and the y^t being set to $M(x, r^1 r^2 \ldots r^t)$).

Definition 3.3.1 We say that the Arthur strategy A defines an Arthur-Merlin proof system for L if the following conditions hold:

(1) *Completeness:* There exists a Merlin strategy M such that $\Pr[\,(A, M) \text{ accepts } w\,] \geq \frac{2}{3}$ for every $w \in L$.

(2) *Soundness:* $\Pr[\,(A, \widehat{M}) \text{ accepts } w\,] \leq \frac{1}{3}$ for every Merlin strategy $\widehat{M}$ and every $w \notin L$.

The strategy $\widehat{M}$ in the soundness conditions is sometimes called a *cheating* Merlin, while the strategy M in the completeness condition is called the *honest* Merlin. In fact, it suffices to consider (in both conditions) an "optimal Merlin", M_{opt_A}, that chooses all its messages in a way maximizing Arthur's accepting probability. Note that M_{opt_A} depends on A.

We define A's *accepting probability function* on partial conversations as follows (cf. [5],[6]):

- $acc(w, r^1 y^1 \ldots r^g y^g) = \rho(w, r^1 y^1 \ldots r^g y^g)$
- $acc(w, r^1 y^1 \ldots r^t y^t r^{t+1}) = \max_y acc(w, r^1 y^1 \ldots r^t y^t r^{t+1}.y)$ for $t = g(n) - 1, \ldots, 0$
- $acc(w, r^1 y^1 \ldots r^t y^t) = E_r acc(w, r^1 y^1 \ldots r^t y^t.r)$ for $t = g(n) - 1, \ldots, 0$.

The following

	subgame 1	subgame 2	...	subgame m	
Arthur's message:	r_1^1	r_2^1	...	r_m^1	} g rounds
Merlin's response:	y_1^1	y_2^1		y_m^1	
$\vdots$	$\vdots$	$\vdots$		$\vdots$	
Arthur's message:	r_1^g	r_2^g		r_m^g	
Merlin's response:	y_1^g	y_2^g	...	y_m^g	

Figure 3-1: Framework of the Standard Error-Reduction Protocol

Proposition 3.3.2 *For any fixed history $r^1y^1 \ldots r^ty^t$ one has*

$$\text{acc}(w, r^1y^1 \ldots r^{t-1}y^{t-1}.r^ty^t) \leq \text{acc}(w, r^1y^1 \ldots r^{t-1}y^{t-1}.r^t)$$

with equality holding if $y^j = M_{\text{opt}_A}(r^1 \ldots r^j)$ for each $j = 1, \ldots, t$.

(which we will use later) is just a restatement of the definition.

A's accepting probability on input w is $acc(w) \stackrel{\text{def}}{=} acc(w, \lambda)$. The *error probability* of A on input w (with respect to a language L) is defined as

$$err_L(w) = \begin{cases} 1 - acc(w) & \text{if } w \in L \\ acc(w) & \text{otherwise.} \end{cases}$$

The error probability of A (with respect to L) is $e_L : \mathbf{N} \to [0,1]$ defined by $e_L(n) = \sup_{|w|=n} err_L(w)$. (Thus an Arthur strategy A defines a proof system for L if $e_L \leq \frac{1}{3}$).

3.3.2 Error-Reduction and its Standard Implementation

Error-reduction is the process of reducing the error probability of an Arthur-Merlin proof system from $\frac{1}{3}$ to 2^{-z} for a given $z = z(n) \leq n^{O(1)}$. We review the standard method of error-reduction [5],[6].

Given $A = (\rho, g, l, q)$ defining an error $\leq \frac{1}{3}$ Arthur-Merlin proof system for L we want to design A^* defining an error $\leq 2^{-z}$ Arthur-Merlin proof system for L. The solution is to play in parallel $m = O(z)$ *independent* copies of the old game (the one defined by strategy A). The independence of Arthur's moves in the various "subgames" is used to prove that the error probability decreases exponentially with the number of subgames.

More concretely, A^* will, in round t, send ml random bits to Merlin. These bits are regarded

	subgame 1	subgame 2	...	subgame m	
Arthur's message s^1 specifies:	r_1^1	r_2^1	...	r_m^1	} g rounds
Merlin's response:	y_1^1	y_2^1		y_m^1	
$\vdots$	$\vdots$	$\vdots$		$\vdots$	
Arthur's message s^g specifies:	r_1^g	r_2^g		r_m^g	
Merlin's response:	y_1^g	y_2^g	...	y_m^g	

Figure 3-2: Framework of Our Error-Reduction Protocol

as a sequence $r_1^t \ldots r_m^t$ of m different round t messages of A. Merlin then responds with strings $y_1^t \ldots y_m^t$, and y_i^t is regarded as the response of Merlin to r_i^t in the i-th subgame ($i = 1, \ldots, m$). This continues for g rounds (see Figure 3-1).

Finally, A^* will accept in the new game iff a majority of the subgames were accepting for the original A. That is, A^* accepts iff $|\{\, i : \rho(w, r_i^1 y_i^1 \ldots r_i^{g(n)} y_i^{g(n)}) = 1 \,\}| \geq \frac{m(n)}{2}$.

The bound on the error probability of the new game follows from the fact that the coin tosses used by Arthur in the different subgames are independent. However, the cost of this argument is in the large number of coin tosses used by A^*; namely $O(lz)$ coin tosses per round (to be contrasted with the l coin tosses used in each round of the original game).

3.3.3 Overview of Our Protocol

We will run m subgames in parallel (with m appropriately chosen). In each round t Arthur sends a random seed s^t of a universal sampler G (whose parameters we will specify later). This specifies a sequence $r_1^t \ldots r_m^t$ of messages that will play the role of A's t-th round messages. Although the same pseudo-random process is used at each round t, it will be with a completely new random seed s^t. At the end, A^* will accept iff a majority of the subgames were accepting (see Figure 3-2).

We emphasize that Arthur sends a seed s^t and both parties then compute the sequence of messages by running the sampler with s^t as coin tosses.

The difficulty now is that a cheating Merlin might be able to capitalize on the dependency between the subgames. That is, although an honest Merlin would compute y_i^t based only on $r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}$ (like the honest Merlin for the original protocol) a cheating Merlin could

compute the string $y_1^t \dots y_m^t$ based on the entire submatrix above this string. Clearly we cannot prevent Merlin from following such a strategy. Using the properties of the universal sampler however, we can show that no such strategy would help.

We will guarantee that at each round the average accepting probability of the m subgames on the sequence specified by the seed approximates the average accepting probability of a sequence of independently chosen messages. That is, assuming $s^1, \dots, s^{t-1}$ specifying $r_1^1 \dots r_m^1, \dots, r_1^{t-1} \dots r_m^{t-1}$ have been chosen, we guarantee that with high probability we have

$$\frac{1}{m}\sum_{i=1}^{m} acc(w, r_i^1 y_i^1 \dots r_i^{t-1} y_i^{t-1} . r_i^t) \approx \frac{1}{m}\sum_{i=1}^{m} E_r\, acc(w, r_i^1 y_i^1 \dots r_i^{t-1} y_i^{t-1} . r)$$

for the random choice of s^t (where $r_1^t \dots r_m^t$ is the sequence specified by s^t). If all seeds selected provide good approximations in this sense then the rate of accepting subgames (in the new game) will approximate the accepting probability (in the original game). Hence, it all amounts to selecting a universal sampler which guarantees that all g approximations are "good" with very high probability.

3.3.4 The Universal Sampler for Error-Reduction

The sampler which we will use to generate the messages at each round is specified by the following

Theorem 3.3.3 *Let $g, l, z : \mathbf{N} \to \mathbf{N}$ be $\leq n^{O(1)}$ with $\log n = o(l)$. Then we can construct a universal $\left(l, \frac{1}{6g}, \frac{2^{-z}}{g}\right)$-sampler which uses $O(l + z \log l)$ coin tosses.*

Proof Apply Theorem 3.2.5 with $\epsilon = \frac{1}{6g}$, $\delta_1 = \frac{1}{g}$, and $\delta_2 = 2^{-z}$. □

3.3.5 Randomness-Efficient Error-Reduction Theorem

Theorem 3.3.4 *Suppose $A = (\rho, g, l, q)$ is an Arthur strategy that has error probability $\leq \frac{1}{3}$ with respect to L. Then we can construct an Arthur strategy $A^* = (\rho^*, g, O(l + z \log l), q^*)$ which has error probability $\leq 2^{-z}$ with respect to L.*

We distinguish two cases. The first is when $l = O(\log n)$ for which we may prove the statement of Theorem 3.3.4 using just the simple universal sampler of Section 3.2.4. We omit that proof and proceed to the more interesting case of $\log n = o(l)$.

Let G be the universal $\left(l, \frac{1}{6g}, \frac{2^{-z}}{g}\right)$-sampler specified by Theorem 3.3.3. Let m be the number of sample points that it outputs and $s = O(l + z \log l)$ the number of random bits it uses. The new Arthur strategy is $A^* = (\rho^*, g, s, mq)$ where $\rho^*(w, s^1 y_1^1 \ldots y_{m(n)}^1 \cdots \cdots s^{g(n)} y_1^{g(n)} \ldots y_{m(n)}^{g(n)})$

$$= \begin{cases} 1 & \text{if } |\{\, i : \rho(w, G_i(s^1) y_i^1 \ldots G_i(s^{g(n)}) y_i^{g(n)}) = 1 \,\}| \geq \frac{m(n)}{2} \\ 0 & \text{otherwise} \end{cases}$$

($n = |w|$ and $G_i(s^t)$ denotes the i-th coordinate of the output of G run with coin tosses $s^t \in \{0,1\}^{s(n)}$).

For the analysis, let $y_1^t \ldots y_m^t$ be Merlin's response in round t.

Proposition 3.3.5 *For each fixed history $s^1.y_1^1 \ldots y_m^1 \cdots s^{t-1}.y_1^{t-1} \ldots y_m^{t-1}$ we have*

$$\Pr\left[\left|\tfrac{1}{m}\textstyle\sum_{i=1}^m \mathrm{acc}(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}.r_i^t) - \mathrm{acc}(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1})\right| < \tfrac{1}{6g}\right] \geq 1 - \tfrac{2^{-z}}{g}$$

where $r_1^j \ldots r_m^j$ is the sequence specified by s^j and the probability is over the random choice of s^t.

Proof The sampler guarantees that

$$\Pr\left[\left|\tfrac{1}{m}\textstyle\sum_{i=1}^m acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}.r_i^t) - E_r acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}.r)\right| < \tfrac{1}{6g}\right] \geq 1 - \tfrac{2^{-z}}{g}\,.$$

But by definition of the accepting probability function (see Section 3.3.1) we know that

$$E_r acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}.r) \;=\; acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1})\,,$$

and the Proposition follows. □

The proof is completed by considering separately the cases of $w \in L$ and $w \notin L$.

Claim 3.3.6 *Suppose $w \in L$. Then there exists a Merlin strategy for which A^* accepts with probability $\geq 1 - 2^{-z}$.*

Proof We choose the particular strategy of setting $y_i^t = M_{\mathrm{opt}_A}(r_i^1 \ldots r_i^t)$ where as usual $r_1^j \ldots r_m^j$ is the sequence specified by s^j. By Proposition 3.3.2 we know that

$$acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}.r_i^t) = acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1}.r_i^t y_i^t)\,.$$

By t applications of Proposition 3.3.5 it follows that at the end of t rounds we have

$$\Pr\left[\tfrac{1}{m}\textstyle\sum_{i=1}^m acc(w, r_i^1 y_i^1 \ldots r_i^t y_i^t) > acc(w) - \tfrac{t}{6g}\right] \geq 1 - \tfrac{t2^{-z}}{g}\,.$$

Thus at the conclusion of the game ($t = g$) we are guaranteed that

$$\Pr\left[\frac{1}{m}\sum_{i=1}^{m} acc(w, r_i^1 y_i^1 \ldots r_i^g y_i^g) > acc(w) - \frac{1}{6}\right] \geq 1 - 2^{-z}.$$

But $\frac{1}{m}\sum_{i=1}^{m} acc(w, r_i^1 y_i^1 \ldots r_i^g y_i^g)$ is just the fraction of accepting subgames. Since $acc(w) \geq \frac{2}{3}$ we conclude that with probability $\geq 1 - \epsilon$ a majority of the subgames accept. □

Claim 3.3.7 *Suppose $w \notin L$. Then A^* accepts with probability $\leq 2^{-z}$.*

Proof By Proposition 3.3.2 we know that

$$acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1} . r_i^t y_i^t) \leq acc(w, r_i^1 y_i^1 \ldots r_i^{t-1} y_i^{t-1} . r_i^t).$$

By t applications of Proposition 3.3.5 it follows that at the end of t rounds we have

$$\Pr\left[\frac{1}{m}\sum_{i=1}^{m} acc(w, r_i^1 y_i^1 \ldots r_i^t y_i^t) < acc(w) + \frac{t}{6g}\right] \geq 1 - \frac{t2^{-z}}{g}.$$

Thus at the conclusion of the game ($t = g$) we are guaranteed that

$$\Pr\left[\frac{1}{m}\sum_{i=1}^{m} acc(w, r_i^1 y_i^1 \ldots r_i^g y_i^g) < acc(w) + \frac{1}{6}\right] \geq 1 - 2^{-z}$$

and the conclusion follows from the fact that $acc(w) \leq \frac{1}{3}$. □

Goldreich [29] has recently improved this protocol to use only $O(l + z)$ random bits per round using his block sampling techniques mentioned in Section 3.1.2.

Chapter 4

One-Way Functions are Necessary and Sufficient for Secure Signatures

4.1 Introduction

In theoretical cryptography, we are concerned with encryption schemes, or other cryptographic primitives such as *digital signatures*, which are provably secure under certain assumptions. Diffie and Hellman first suggested in [26] that modern cryptography be based on the assumption that one-way functions (which are easy to compute, but hard to invert) exist. Stronger assumptions have been considered; for example, the existence of a trapdoor permutation (a one-way permutation which is easy to invert given an associated secret) allows the construction of a *public-key encryption scheme.* Much research in theoretical cryptography has been centered around finding the weakest possible cryptographic assumptions required to implement major primitives. For example, pseudo-random generators at first could only be constructed from a specific hard problem, such as discrete log [17]. Later it was shown how to construct pseudo-random generators given any one-way permutation [66], and from other weak forms of one-way functions [46, 30]. Finally it was proved in [36, 34] that the existence of any one-way function was a necessary and sufficient condition for the existence of pseudo-random generators. Similarly, the existence of trapdoor permutations can be shown to be necessary and sufficient for secure encryption schemes.

Progress on characterizing the requirements for secure digital signatures has been slower in

coming, however. We will be interested in signature schemes which are secure against existential forgery under adaptive chosen message attacks. This notion of security, as well as the first construction of digital signatures secure in this sense was provided by Goldwasser, Micali, and Rivest in [32]. Their scheme was based on factoring, or more generally, the existence of claw-free pairs. Bellare and Micali [11] showed how to construct secure signatures based on any trapdoor permutation. This was recently extended by Naor and Yung [55] to only require a one-way permutation. In this paper, we present a method for constructing secure digital signatures given any one-way function. This is the best possible result, since a one-way function can be constructed from any secure signature scheme.

Our method follows [55] in basing signatures on families of one-way hash functions: functions which compress their input, but have the property that even given one pre-image, it is hard to find a different one. The image of a message under one of these functions in itself provides a weak form of signature; [55] shows how to build the stronger secure signatures from this primitive. To complete their construction, they provide a simple method for constructing a one-way hash function from any one-way permutation.

However, arbitrary one-way functions must be managed much more carefully. The bulk of this paper is concerned with building a one-way hash function given any one-way function. First we show how to build a function which has the property that given one pre-image, although most other pre-images may be trivial to find, a small fraction must be hard. Then we show how to amplify this into a full one-way hash function. Our proof makes heavy usage of universal hash functions [19] (see Section 3.2.2) and the k-wise independence tail inequality (Lemma 2.2.10).

4.2 Preliminaries

4.2.1 Notation

Function $f : \{0,1\}^n \rightarrow \{0,1\}^m$ is one-way in the uniform (non-uniform) model if f is polynomial-time computable, but there is not a probabilistic polynomial-time algorithm A (polynomial-sized family of circuits A) such that

$$\Pr[f(A(f(x), 1^n)) = f(x)] > n^{-c}$$

for infinitely many n, where the probability is as x is chosen uniformly from $\{0,1\}^n$. In this paper, we will be assuming the uniform model of security unless stated otherwise. Our result that one-way functions are necessary and sufficient for secure signatures holds in the non-uniform model as well.

Given a function f, we define the *siblings* of x under f, $S_f(x)$, to be the elements which f maps to $f(x)$, i.e.

$$S_f(x) = f^{-1}(\{f(x)\}) = \{x' : f(x') = f(x)\}.$$

It will be useful to partition the domain of f into subsets having approximately the same number of siblings. For this purpose, we define the i^{th} segment of the domain of f to be,

$$D_i(f) = \{x : 2^i \leq |S_f(x)| < 2^{i+1}\}.$$

Corresponding to our partition of the domain of f, we partition the range of f according to the number of pre-images, and define the i^{th} segment of the range of f to be,

$$R_i(f) = f(D_i(f)) = \{y : 2^i \leq |f^{-1}(y)| < 2^{i+1}\}.$$

We will denote the concatenation of strings x and y by $x \cdot y$. We will denote the composition of functions f and g by $f \circ g$, i.e. $(f \circ g)(x) = f(g(x))$. Given functions $f : \{0,1\}^{n_1} \rightarrow \{0,1\}^{n_2}$ and $g : \{0,1\}^{n_3} \rightarrow \{0,1\}^{n_4}$, we define $f \cdot g : \{0,1\}^{n_1+n_3} \rightarrow \{0,1\}^{n_2+n_4}$ to be the function which maps $x \cdot y$ to $f(x) \cdot g(y)$.

Given families of functions F and G, we define $F \circ G = \{f \circ g : f \in F, g \in G\}$, $F \cdot G = \{f \cdot g : f \in F, g \in G\}$, and $F^n = F \cdot F \cdot \ldots \cdot F$ (n times).

4.2.2 Signature Schemes

A signature scheme has the following components:

- A security parameter n, determining the security, running time, and lengths of messages.
- A message space, which we will assume to be strings in $\{0,1\}^n$.
- A polynomial S_B called the signature bound

- A probabilistic polynomial-time key generation algorithm KG, which on input 1^n, outputs a public key PK, and a matching secret key SK.
- A probabilistic polynomial-time signing algorithm SP, which given a message m and a matching pair of keys <PK,SK>, outputs a signature of m with respect to PK.
- A polynomial-time verification algorithm V, which given S, m, and PK, tests whether or not S is a valid signature of m with respect to PK.

The notion of security we are interested in is called security against existential forgery under adaptive chosen message attack. This means we will allow our adversary, the forger, to adaptively choose messages, and be supplied with valid signatures for these messages. We will consider him successful if, on his own, he is able to produce a valid signature for any message that we did not sign for him. We will be interested in signature schemes for which no polynomial-sized forger has a $1/p(k)$ chance of producing a forged message, for any polynomial p and for sufficiently large k. This is the strongest natural notion of security (see [32] or [11] for a more complete discussion of the model).

We note that many of the signature schemes originally devised are not secure in this strong manner. For example, the signature scheme proposed by Rivest, Shamir, and Adleman [58] is based on the assumption that taking cube roots modulo a large composite number N is hard without knowing the factorization of N. Their scheme has N as the public key and the factorization of N as the private key. They then propose using $\sqrt[3]{m} \pmod N$ as the signature of message m (which is assumed to be an integer between 1 and $N-1$). However, a forger can make use of algebraic properties, such as $\sqrt[3]{m_1 m_2} \equiv \sqrt[3]{m_1}\sqrt[3]{m_2} \pmod N$, to break this scheme under a chosen message attack. For example, to forge a signature for message m, the forger need only obtain signatures for mr and r^{-1}, where r is an arbitrary element of the multiplicative group modulo N, and then multiply the two signatures to obtain a valid signature for m. Other schemes fail because of similar algebraic relations. The notion of security against existential forgery under chosen message attack was, in fact, for some time considered paradoxical. The first signature scheme achieving this notion of security was obtained by Goldwasser, Micali, and Rivest [32].

Before showing sufficient conditions for secure signature schemes to exist, we will first give a necessary condition due to Bellare [9].

Lemma 4.2.1 *The existence of a secure signature scheme implies the existence of a one-way function.*

Proof Let $f(1^n, x)$ run the KG algorithm on input 1^n and using random tape x, and output PK. Then f is a one-way function. Why? Assume we could invert f. Then given the public key PK, we would be able to obtain a secret key SK′ with the property that SK′ could generate signatures valid for PK. But this implies that the signature scheme is insecure, which is a contradiction. □

The remainder of this chapter will be concerned with proving that the existence of a one-way function is sufficient to implement secure signatures

The signature schemes of [11] and [55] both use the notion of a window, due to Diffie and Lamport [26, 44]. In this limited signature scheme, the signer randomly picks some one-way function f and $2l$ strings $\alpha_1^0, \alpha_1^1, \ldots, \alpha_l^0, \alpha_l^1$ from the domain of f. Next, he computes $\beta_i^j = f(\alpha_i^j)$ for $1 \leq i \leq l$ and $j \in \{0,1\}$. The signer then places in the public file f and $\beta_1^0, \beta_1^1, \ldots, \beta_m^0, \beta_m^1$. To sign a message $m = m_1 m_2 \cdots m_l \in \{0,1\}^l$, the signer reveals $\alpha_i^{m_i}$ for $1 \leq i \leq l$. To verify a signature, one need only verify that $f(\alpha_i^{m_i}) = \beta_i^{m_i}$, for $1 \leq i \leq l$. Assume that, after getting a message m of his choice signed for him, a forger could sign a different message m'. For some i, $m_i' \neq m_i$; for that i, the forger must invert f on $\beta_i^{m_i'}$, which is impossible, since f is one-way.

The limitation of the above scheme is that only l bits can be signed. Bellare and Micali [11] devised a way to regenerate the public file, and thus sign an arbitrary polynomial number of messages. The scheme in [11] is to have f be a randomly chosen trapdoor permutation. At each stage, they sign a message and a description of a new f using the Diffie-Lamport scheme. In this way, the same β's could be used over and over. The signature which is output includes a complete history of all functions f used. To verify the signature, one uses the f in the public file to verify the second f, then uses the second f to verify the third f, etc.; the last f is used to verify the message. This can be improved by at each step signing a message and two new f's. Then a binary tree of signatures can be formed, making the length of the signature grow logarithmically in the number of messages rather than linearly in the number of messages.

Naor and Yung [55] use a variant of this scheme, strongly resembling a practical scheme proposed by Merkle [50, 51, 52]. The idea is, instead of signing new f's, they seek to sign a new set of β's. This cannot be done directly, since the Diffie-Lamport signature scheme increases the length of the message by a polynomial factor. Instead, Merkle and Naor-Yung use the Diffie-Lamport scheme to sign the hash value of a new $\{\beta_i^j\}$ table. However this signature scheme now allows two paths to forgery: inverting f on some β_i^j or finding a table $\{\gamma_i^j\}$ with known inverses and the same image under the hash function as the real $\{\beta_i^j\}$ table. Thus, to prevent this second attack, the hash function must itself have cryptographic properties. Roughly what is required is that given x and $h(x)$, one cannot compute an $x' \neq x$ such that $h(x') = h(x)$. We call such an x' a *non-trivial sibling* of x under h.

To make this notion more formal, Naor and Yung define a family of one-way hash functions to be a collection of polynomial-time computable functions $G = \{g_i\}$ from n bits to n' bits ($n > n'$) such that there is no *non-trivial sibling finder*, i.e. no probabilistic polynomial-time algorithm which initially chooses $x \in \{0,1\}^n$, then gets a random $g \in G$, and outputs a non-trivial sibling x' of x under g with probability at least n^{-c} over its random coins and the choice of g. They show that a secure signature scheme can be constructed from a family of one-way hash functions together with any one-way function.

They then show how to construct a family of one-way hash functions given any one-way permutation $f : \{0,1\}^n \to \{0,1\}^n$ as follows. Let $H = \{h_i\}$ be a 2-universal family of hash functions from n bits to $n-1$ bits. They then let $g_i = h_i \circ f$.

Lemma 4.2.2 ([55]) *$G = \{g_i\}$ is a family of one-way hash functions.*

Proof For convenience we will assume that the h_i map exactly 2-1 (this is not required, but such families of hash functions do exist). Assume there exists a non-trivial sibling finder A for G. We will use A to create a probabilistic polynomial-time algorithm B to invert f. Assume B is given a random $z \in \{0,1\}^n$. Algorithm B will begin by getting x from A. In the unlikely event that $f(x) = z$ then it will output x. Otherwise, it will then pick $h \in H$ at random subject to the constraint that $h(z) = h(f(x))$ and give $g = h \circ f$ to A. Finally, it will get x' from A and output it.

Note that because H is 2-universal and exactly 2-1, picking a random $z \neq f(x)$ and a random $h \in H$ such that $h(z) = h(f(x))$ is exactly the same as picking a random $h \in H$. Thus

algorithm A, with probability n^{-c}, will output an $x' \neq x$ such that $g(x') = g(x)$. But, since h is 2-1 and $h(z) = h(f(x))$, $x' = f^{-1}(z)$. So B inverts f with probability at least n^{-c}. □

Our proof that secure signatures can be constructed from any one-way function will actually consist of showing that a family of one-way hash functions can be constructed from any one-way function. We will then apply the reduction of [55] to obtain a secure signature scheme.

4.3 Constructing a One-way Hash Function

4.3.1 Overview

In this section we will show how to, given any one-way function, create a family of one-way hash functions. This will imply, by our discussion in the previous section, that we can construct, given any one-way function, a signature scheme secure against existential forgery under adaptive chosen message attacks.

Starting with our original one-way function f, we construct a series of families of functions, each one closer to our goal of a family of one-way hash functions. In Section 4.3.2, we construct a family F_1 with the property that, although most siblings under a random $f_1 \in F_1$ may be easy to find, a non-negligible fraction are provably hard to find. Next, in Section 4.3.3, we construct a family F_2 such that most siblings under most $f_2 \in F_2$ are provably hard to find. Then, in Section 4.3.4, we construct a family F_3 with the property that for almost all $f_3 \in F_3$, it is hard to find any sibling. This is a family of length-increasing functions , so in Section 4.3.5 we construct from it a family F_4 of functions which are sibling-hard and length-decreasing. Finally, in Section 4.3.6, we give a family of one-way hash functions.

4.3.2 Making Some Siblings Hard

The crucial cryptographic property of a family of one-way hash functions is that it is hard to find a sibling of any domain element. Naor and Yung start with a one-way permutation, which trivially has this property, and thus only had to show how to make the function compress as well.

We are not so fortunate. Consider, for example, the function $f(< x, y >) =< g(x), 1^k >$, where x and y are k-bit strings, and g is some one-way permutation. It it easy to show that f

is one-way, but that each element of the domain has an exponential number of siblings, all of which are trivial to find. Nevertheless, in this section, we will show how to take *any* one-way function and convert it into a family of functions F_1 with the property that, for any input, a non-negligible fraction of its siblings under a non-negligible fraction of the $f_1 \in F_1$ are provably hard to find.

We begin by putting our one-way function into a normal form. Assume we are given a function $f : \{0,1\}^m \to \{0,1\}^m$ which is one way. Let $l(m) = \lceil \log m \rceil$ and $k(m) = 2^{l(m)}$. Let

$$n = m + 4k(m) + l(m) + 2.$$

We construct the function $f_0 : \{0,1\}^n \to \{0,1\}^n$ such that for $x \in \{0,1\}^m$, $y \in \{0,1\}^{4k(m)}$, and $i \in \{0,1\}^{\log 4k(m)}$,

$$f_0(x \cdot y \cdot i) = f(x) \cdot (y \wedge (1^i \cdot 0^{4k(m)-i})) \cdot i,$$

where the $\wedge$ is bitwise AND. The reason for constructing f_0 is that its pre-image structure is largely known.

Lemma 4.3.1 *$|D_j(f_0)|$, as a function of j, is increasing in the range 0 to m, flat and equal to $2^n/4k(m)$ in the range m to $4k(m)$, and decreasing in the range $4k(m)$ to $m + 4k(m)$.*

Proof

$$\begin{aligned} |D_j(f_0)| &= \sum_{i=0}^{4k(m)-1} 2^{4k(m)} |D_{j-i}(f)| \\ &= 2^{4k(m)} \sum_{i=\max(0,j-4k(m)+1)}^{\min(m,j)} |D_i(f)|. \end{aligned}$$

From this form, one can easily observe the monotone nature of $|D_j(f_0)|$ for small and large j. For $m \leq j < 4k(m)$, we note that

$$\begin{aligned} |D_j(f_0)| &= 2^{4k(m)} \sum_{i=0}^{m} |D_i(f)| \\ &= 2^{4k(m)+m} \\ &= 2^n/4k(m). \square \end{aligned}$$

However, f_0 does not lose the "one-wayness" of f.

Lemma 4.3.2 *If f is a one-way function, then f_0 is a one-way function.*

Now we can construct a family F_1 such that a non-negligible fraction of the siblings are hard to find. We let $F_1 = H^n_{(n/2+\log n),n} \circ \{f_0\} \circ H^n_{n,(n/2-2\log n)}$, i.e. to obtain a random $f_1 \in F_1$ we randomly choose h_1 from an n-universal family of hash functions mapping $(n/2+\log n)$-bit strings to n-bit strings, randomly choose h_2 from an n-universal family of hash functions mapping n-bit strings to $(n/2-2\log n)$-bit strings, and let $f_1 = h_2 \circ f_0 \circ h_1$ (f_1 maps $\{0,1\}^{n/2+\log n}$ to $\{0,1\}^{n/2-2\log n}$).

Next we must define what we mean by a *hard sibling*. For any $f_1 = h_2 \circ f_0 \circ h_1 \in F_1$, and for any $x \in \{0,1\}^{n/2-2\log n}$, we define the hard sibling set $H_{f_1}(x)$ to be the set of y's in $\{0,1\}^{n/2-2\log n}$ such that

1. $f_1(y) = f_1(x)$
2. $f_0(h_1(y)) \neq f_0(h_1(x))$
3. $h_1(y) \in D_{n/2}(f_0)$.

The first condition states that y is a sibling of x. We will not be able to prove any hardness results about elements which collide with x on h_1 or f_0, so we will simply declare them to be easy siblings. The second condition states that y is a sibling of x due to a collision on h_2, not on f_0 (or h_1). We give for technical reasons which will become apparent below, the third condition on a sibling being hard. For convenience, we also define the set of easy siblings, $E_{f_1}(x) = S_{f_1}(x) - H_{f_1}(x)$.

Now we can give two lemmas which make precise the hardness of f_1. The first is that finding hard siblings is actually difficult. First let us define what it means for an algorithm to find hard siblings.

Definition 4.3.3 A *hard sibling finder* for F_1 is probabilistic polynomial-time algorithm A which chooses an $x \in \{0,1\}^{n/2+\log n}$, gets a random $f_1 \in F_1$, and with probability at least n^{-c} over its random coins and the choice of f_1 outputs a $y \in H_{f_1}(x)$.

Lemma 4.3.4 *If there exists a hard sibling finder A for F_1, then there exists a probabilistic polynomial-time algorithm B which inverts f_0 with non-negligible probability.*

Proof Algorithm B, given $z = f_0(w)$ (where we assume w is picked uniformly from $\{0,1\}^n$), gets x from A, picks h_1 at random, picks $r \in \{0,1\}^{n/2-2\log n}$ at random, then picks h_2 at random subject to the constraint that $h_2(f_0(h_1(x))) = h_2(z) = r$. Algorithm B then gives $f_1 = h_2 \circ f_0 \circ h_1$ to A. It then lets y be the output of A. Finally outputs $h_1(y)$.

We will, in fact, only try to invert strings in $R_{n/2}(f_0)$. So, for now, let us assume that w is picked uniformly from $D_{n/2}(f_0)$. Also, assume x is chosen by A and that the probability that A will subsequently output an element of $H_{f_1}(x)$ assuming a random f_1 is p_x. Under these assumptions, let us first consider the probability that A outputs an element of $H_{f_1}(x)$ when f_1 is supplied by B. This probability can be written as

$$\sum_{h_1,h_2} \Pr[A_n \text{ outputs } y \in H_{f_1}(x) | h_1, h_2] \Pr[B \text{ picks } h_1, h_2].$$

The above summation would be p_x if the probability that B selected h_1 and h_2 was exactly equal to the probability of picking h_1 and h_2 uniformly at random. What we will in fact show is that, for almost all h_1 and h_2, these two probabilities differ by at most a $(1 + O(1/\sqrt{n}))$ factor. This implies that the summation above is at least $p_x(1 - O(1/\sqrt{n}))$.

Fix any h_1 and r. Let $G(h_2)$ contain the elements of $D_{n/2}(f_0)$ which $h_2 \circ f_0$ maps to r, i.e.

$$G(h_2) = f_0^{-1}(h_2^{-1}(r)) \cap D_{n/2}(f_0).$$

The expected size of $G(h_2)$ is $n^2 2^{n/2}/4k(m)$. We can also write $|G(h_2)|$ as

$$2^{n/2+1} \sum_{y \in R_{n/2}(f_0)} X_y,$$

where X_y takes on value $2^{-n/2-1}|f_0^{-1}(y)|$ if $h_2(y) = r$ and 0 otherwise. Each X_y takes on values between 0 and 1, and since h_2 is randomly chosen from an n-universal family of hash functions, the X_y's are n-wise independent. Thus we can apply Lemma 2.2.10 with $k = \sqrt{n}$ to show that

$$\Pr[\sum_y X_y - E[\sum_y X_y] > 2n^{3/4}] < 2^{-\sqrt{n}}.$$

So for an overwhelming fraction of the h_2's

$$(\frac{n^2}{4k(m)} - n^{3/4})2^{n/2} \leq |G(h_2)| \leq (\frac{n^2}{4k(m)} + n^{3/4})2^{n/2}.$$

This holds even if we first restrict to those h_2 which map $f_0(h_1(x))$ to r, since even with that restriction the X_i's are $(n-1)$-wise independent (and we applied Lemma 2.2.10 assuming only

$\sqrt{n}$-wise independence). Since for every w, the fraction of the h_2's for which $h_2(f_0(w)) = h_2(f_0(h_1(x))) = r$ is exactly $2^{-n-4\log n}$, we can think of B as uniformly picking a pair $< w, h_2 >$ from the set $\{< w, h_2 > | h_2(f_0(w)) = h_2(f_0(h_1(x))) = r\}$. So clearly, the probability of picking h_2 is exactly proportional to the size of $|G(h_2)|$, and is thus within a $(1 + O(1/\sqrt{n}))$ factor of uniform for almost all h_2.

Now let us assume that A does in fact output a hard sibling y. Then in particular, $h_1(y) \in G(h_2) - S_{f_0}(h_1(x))$. But given A's view of the computation (i.e. h_1, h_2, r, and x), w is just a random element of $G(h_2) - S_{f_0}(h_1(x))$. So

$$\begin{aligned} & \Pr[f_0(h_1(y)) = f_0(w) = z] \\ & \quad \frac{|S_{f_0}(w)|}{|G(h_2)|} \\ & \geq \frac{4k(m)}{n^2} + O(n^{3/4}) \\ & \geq 1/n. \end{aligned}$$

So if w is picked uniformly from $D_{n/2}(f_0)$ then the probability B outputs a pre-image of z is at least $1/n$. Since $D_{n/2}(f_0)$ accounts for more than a $1/n$ fraction of $\{0,1\}^n$, this implies that the probability the B outputs a pre-image of z when w is picked uniformly from $\{0,1\}^n$ is at least $1/n^2$, given that A succeeds in outputting a hard sibling. Thus we have,

$$\begin{aligned} \Pr[B \text{ inverts } z] & \geq n^{-2} \sum_x \Pr[A \text{ initially outputs } x] p_x \\ & \geq n^{-2}(n^{-c}(1 - O(1/\sqrt{n})) - O(2^{-\sqrt{n}})) \\ & \geq \Omega(n^{-c-2}). \quad \square \end{aligned}$$

Lemma 4.3.4 says that we cannot find hard siblings in polynomial time. The next lemma says that these hard siblings for a given w form a non-negligible fraction of the possible siblings of w.

Lemma 4.3.5 *Fix $w \in \{0,1\}^{n/2+\log n}$ and let X be a random variable (dependent on the random choice of f_1) such that*

$$X = \log \frac{|S_{f_1}(w)|}{|E_{f_1}(w)|},$$

i.e. X is the logarithm of the ratio between the number of siblings and the number of easy siblings. Then the X takes on values in the range 0 to n and has expected value at least $\Omega(1/n)$.

Proof Consider some $w \in \{0,1\}^{n/2+\log n}$. We will show that with constant probability, w has at most n^3 siblings, and at least n^2 of them are hard. Pick $x \in \{0,1\}^n$ and $z \in \{0,1\}^{n/2-2\log n}$ at random. We will restrict our attention to hash functions h_1 which map w to x and h_2 which map $y = f_0(x)$ to z. Note that picking x at random and then h_1 at random subject to $h_1(w) = x$ is the same as just picking h_1 at random; similarly, picking z at random and then picking h_2 at random subject to $h_2(y) = z$ is the same as just picking h_2 at random.

To count the number of siblings w' of w under f_1, we first count the number of siblings x' of x under $h_2 \circ f_0$, and then count the number of w' that h_1 maps to such x'.

Let y be in the range of f_0. We let X_y take on value $|f_0^{-1}(y)|$ if $h_2(y) = z$, and 0 otherwise. Then, the number of siblings x' of x under $h_2 \circ f_0$ can be written as,

$$\begin{aligned}
|S_{h_2 \circ f_0}(x)| &= \sum_{y \in \text{Range}(f_0)} X_y \\
&\leq \sum_{i<\frac{n}{2}} \sum_{y \in R_i(f_0)} X_y + \sum_{\frac{n}{2} \leq i < \frac{n}{2}+\frac{3}{2}\log n} \sum_{y \in R_i(f_0)} X_y + \\
&\quad \sum_{i \geq \frac{n}{2}+\frac{3}{2}\log n} \sum_{y \in R_i(f_0) - \{f_0(x)\}} X_y + |S_{f_0}(x)|
\end{aligned}$$

We bound each of these four terms separately. Letting $X'_y = 2^{-n/2} X_y$, we get

$$\sum_{i<\frac{n}{2}} \sum_{y \in R_i(f_0)} X_y = 2^{-n/2} \sum_{i<\frac{n}{2}} \sum_{y \in R_i(f_0)} X'_y.$$

Each X'_y in the above sum takes on values between 0 and 1. Furthermore, by Lemma 4.3.1, and the fact that, for any y, the probability that $h_2(y) = z$ is $2^{-n/2+2\log n}$,

$$\begin{aligned}
E[\sum_{i<\frac{n}{2}} \sum_{y \in R_i(f_0)} X'_y] &= 2^{-n+2\log n} \sum_{i<\frac{n}{2}} |D_i(f_0)| \\
&\leq \frac{5}{8} n^2.
\end{aligned}$$

Thus, by Lemma 2.2.10 (using $k = n$),

$$\Pr[\sum_{i<\frac{n}{2}} \sum_{y \in R_i(f_0)} X'_y > \frac{5}{8} n^2 + 2n^{7/4}] < 2^{-n}.$$

So,

$$\Pr[\sum_{i<\frac{n}{2}} \sum_{y\in R_i(f_0)} X_y > (\frac{5}{8}n^2 + O(n^{7/4}))2^{n/2}] < 2^{-n}.$$

By a similar argument, we obtain,

$$\Pr[\sum_{\frac{n}{2}\le i<\frac{n}{2}+\frac{3}{2}\log n} \sum_{y\in R_i(f_0)} X_y > 4n^{7/4}n^{n/2}] \le 2^{-\sqrt{n}}.$$

To bound the third term, we note that,

$$\begin{aligned}\Pr[\sum_{i\ge\frac{n}{2}+\frac{3}{2}\log n} \sum_{y\in R_i(f_0)-\{f_0(x)\}} X_y > 0] &\le 2^{-n/2+2\log n} \sum_{i\ge\frac{n}{2}+\frac{3}{2}\log n} |R_i(f_0)| \\ &\le 4/\sqrt{n}.\end{aligned}$$

Finally, we note, since x was picked at random,

$$\Pr[|S_{f_0}(x)| \ge 2^{n/2}] \le 5/8.$$

Putting the above bounds together, we find that with probability at least $3/8 - O(1/\sqrt{n})$,

$$|S_{h_2\circ f_0}(x)| \le (\frac{5}{8}n^2 + O(n^{7/4}))2^{n/2}.$$

Assume that the above inequality holds, and now consider the number of siblings w' of w under f_1. Note that,

$$S_{f_1}(w) = h_1^{-1}(S_{h_2\circ f_0}(x)).$$

For $w' \in \{0,1\}^{n/2+\log n}$, let X'_w take on value 1 if $h_1(w') \in S_{h_2\circ f_0}(x)$, and 0 otherwise. Then,

$$|S_{f_1}(w)| = 1 + \sum_{w'\ne w} X_w,$$

and thus by Lemma 2.2.10, we can show

$$\Pr[|S_{f_1}(w)| > \frac{2}{3}n^3] < 2^{-n},$$

under the assumption that the previous inequality holds. Thus, with probability at least $3/8 - O(1/\sqrt{n})$,

$$|S_{f_1}(w)| \le \frac{2}{3}n^3.$$

Now consider the number of hard siblings of w. The hard siblings are those w' which h_1 maps to $G(h_2) - f_0^{-1}(y)$. With overwhelming probability, the size of this latter set is at least $1.1n2^{n/2}$, and thus with overwhelming probability, there are at least n^2 hard siblings of w.

Therefore, with probability at least $1/3$, X is at least $1/n$, so $E[X] \ge 1/3n$. □

4.3.3 Making Most Siblings Hard

In the previous section, we showed how to, given any one-way function, construct a family of functions for which some of the siblings are hard to find. In this section we will construct a family of functions for which almost all siblings are hard to find.

Let $F_2 = F_1^{2n^5}$, i.e. a random $f_2 \in F_2$ is constructed to run $2n^5$ different f_1's in parallel. Let $f_2 = f_1^1 \cdot f_1^2 \cdot \ldots \cdot f_1^{2n^5}$.

To define our set of hard siblings, we let

$$E_{f_2}(x^1 \cdot \ldots \cdot x^{2n^5}) = \{y^1 \cdot \ldots \cdot y^{2n^5} | y^i \in E_{f_1^i} \text{ for } 1 \leq i \leq 2n^5\},$$

and let $H_{f_2}(x) = S_{f_2}(x) - E_{f_2}(x)$, i.e. a sibling is easy if and only if it is easy in each component. We can define a notion of a hard sibling finder for F_2 in a way completely analogous Definition 4.3.3.

Again, we must show two facts, first that our notion of hardness is correct, and second that most siblings are in fact hard.

Lemma 4.3.6 *If there is a hard sibling finder A for F_2, then there is a hard sibling finder B for F_1.*

Proof Given an algorithm for finding hard siblings under F_2, we obtain an algorithm for finding a hard siblings under F_1 as follows. Algorithm B will run algorithm A to obtain $x = x_1 \cdot \ldots \cdot x_{2n^5}$. It will then select an i at random between 1 and $2n^5$, output x_i, and attempt to find a hard sibling for x_i. Given f_1^i picked at random, we pick f_1^j for $i \neq j$ at random from F_1 and run A to get $y = y_1 \cdot \ldots \cdot y_{2n^5}$, and output y_i. If y is a hard sibling of x under f_2, then some y_j must be a hard sibling of x_j under f_1. Since i was chosen at random, there is a $1/2n^5$ chance that $i=j$. Thus with probability at least $1/2n^{c+5}$, we output a hard sibling for x_i. □

Lemma 4.3.7 *Fix $x \in \{0,1\}^{n^6+2n^5 \log n}$ and let X be a random variable (dependent on the random choice of f_2) such that*

$$X = \log \frac{|S_{f_2}(x)|}{|E_{f_2}(x)|}.$$

Then the probability the $X \leq \Theta(n^4)$ is at most $e^{-\Theta(n)}$. In other words, with all but an exponentially small probability, there is an exponential gap between the total number of siblings and the number of easy siblings.

Proof Because both easy siblings and all siblings of f_2 are just cross products of the respective sets from f_1, we get that X is the sum of $X_1, \ldots, X_{2n^5}$, chosen independently according to the distribution defined in Lemma 4.3.5. But by Lemma 4.3.5, each X_i is between 0 and n and has expected value at least $1/3n$. The lemma follows from applying Chernoff bounds. □

4.3.4 Making All Siblings Hard

From the previous section, we now have constructed a family of functions F_2 where almost all the siblings under most $f_2 \in F_2$ are hard to find. Unfortunately, there still may be an exponential number of easy siblings for any element of the domain. In this section, we show how to use the exponential gap between easy and hard siblings to construct a family of functions F_3 for which, except for an exponentially small fraction of the functions $f_3 \in F_3$, *all* non-trivial siblings are hard.

Let us look more carefully at Lemma 4.3.7. We rewrite the random variable X as $Y - Z$, where random variable $Y = \log|S_{f_2}(x)|$ and random variable $Z = \log|E_{f_2}(x)|$. Assume for now that we are told the values of $E[Y]$ and $E[Z]$ (we will remove this assumption in Section 4.3.6). Let $l = (E[Y] + E[Z])/2$. Then let $F_{3,l} = F_2 \circ H^2_{(n^6+2n^5\log n-l),(n^6+2n^5\log n)}$. Consider $f_3 = f_2 \circ h_3 \in F_{3,l}$. We can define the hard sibling set of an element under f_3 as

$$H_{f_3}(x) = \{y \in S_{f_3}(x) - S_{h_3}(x) | h_3(y) \in H_{f_2}(h_3(x))\}.$$

Lemma 4.3.8 *Suppose $l > n$. If there exists a hard sibling finder A for $F_{3,l}$ then there exists a hard sibling finder B for F_2.*

Proof Algorithm B will get x from A, and output $h_3(x)$. It will then get a randomly chosen $f_2 \in F_2$, randomly choose $h_3 \in H^2_{(n^6+2n^5\log n-l),(n^6+2n^5\log n)}$, and give $f_3 = f_2 \circ h_3$ to A. It will get y from A and output $h_3(y)$. With probability $1 - 2^{-l} \geq 1 - 2^{-n}$, $S_{h_3}(x) = \emptyset$. Assuming this is the case, $h_3(y) \in H_{f_3}(h_3(x))$ if and only if $y \in H_{f_2}(x)$. So the probability that $h_3(y) \in H_{f_3}(h_3(x))$ is at least $n^{-c} - 2^{-n}$. □

However, we will now show that under most $f_3 \in F_3$, all non-trivial siblings are hard. The intuition is that h_3 selects a very small subset of the domain of f_2. This subspace is so small that it will not contain any easy siblings, although it will contain many hard siblings. This intuition is captured more formally in the following lemma.

Lemma 4.3.9 *Suppose* $l \geq E[Z] + n^4/4$. *Then for any* $x \in \{0,1\}^{n^6+2n^5 \log n - l}$, *with probability at least* $1 - e^{-\Theta(n)}$ *(over the choice of* f_3*).* $S_{f_3}(x) = H_{f_3}(x) \cup \{x\}$.

Proof Assume $x = x_1 \cdot x_2 \cdot \ldots \cdot x_{2n^5}$. Consider a randomly chosen $f_3 = f_2 \circ h_3$, where $f_2 = f_1^1 \cdot f_1^2 \cdot \ldots \cdot f_1^{2n^5}$. Define random variables $Z_1, \ldots, Z_{2n^5}$, where each Z_i is the logarithm of the number of easy siblings of x_i under f_1^i, and is between 0 and n. Also, define random variable Z to be the logarithm of the number of easy siblings of x under f_2. Then, $Z = \sum_i Z_i$, and thus, by Chernoff bounds,

$$\Pr[|Z - E[Z]| > a] < e^{-a^2/4n^7}.$$

In particular, plugging in $a = (l - E[Z])/2 \geq n^4/8$, we get that the probability $Z > (E[Z]+l)/2$ is at most $e^{-n/256}$.

Now consider the probability that x has a non-trivial easy sibling under f_3. The probability that h_3 maps any element other than x to $E_{f_2}(x)$ is at most $2^{n^6+2n^5 \log n - l} \frac{|E_{f_2}(x)|}{2^{n^6+2n^5 \log n}} = |E_{f_2}(x)|/2^l$, which assuming that $Z \leq (E[Z] + l)/2$, is at most $2^{-n^4/8}$. □

Corollary 4.3.10 *If* $l \geq E[Z] + n^4/4$, *then any non-trivial sibling finder for* $F_{3,l}$ *is a hard sibling finder for* $F_{3,l}$.

4.3.5 Compressing

We have finally achieved a function with the hard-sibling property that we want. However, there are still a couple of problems left to be solved. The most obvious is that, in our quest to get the hard-sibling property, we have created a length-increasing function. In particular, the family $F_{3,l}$ constructed in the previous section maps $(n^6 + 2n^5 \log n - l)$-bit strings to $(n^6 - 4n^5 \log n)$-bit strings. Since one can show that $l = \Theta(n^6)$, it is clear that $f_3 \in F_{3,l}$ expands its input. In fact, simply applying a randomly selected hash function h_4 mapping $(n^6 - 4n^5 \log n)$-bit strings to $(n^6 + 2n^5 \log n - l - n/100)$-bit strings to the result of f_3 will solve the problem. So let

$$F_{4,l} = H^2_{(n^6-4n^5 \log n),(n^6+2n^5 \log n - l - n/100)} \circ F_{3,l}.$$

Then we have the following lemma.

Lemma 4.3.11 *Suppose* $l \leq E[Y] - n^4/3$. *Fix* $x \in \{0,1\}^{n^6+2n^5\log n-l}$. *Then, for any* $x \in \{0,1\}^{n^6+2n^5\log n-l}$, *with probability at least* $1-2^{-\Theta(n)}$ *for a randomly chosen* $f_4 = h_4 \circ f_3 \in F_{4,l}$, h_4 *induces no collisions with* x, *i.e.* $S_{f_4}(x) = S_{f_3}(x)$.

Proof We will bound the size of the range of f_3 by $2^{n^6+2n^5\log n-l-n/80}$. Once we have established this fact, the lemma follows trivially. In principle, we would like to bound the range of f_3 by

$$\sum_{i=0}^{n^6+2n^5\log n-l} 2^{-i}|D_i(f_3)|.$$

In fact, it will be more convenient to work with f_2. We will consider 2 cases.

First, consider $z \in \bigcup_{i\geq l} R_i(f_2)$. We will conservatively assume that all such z are in the range of f_3, i.e. if we expect to have z in the range, then just assume it is. To bound the number of such z, we note that

$$\begin{aligned} |\bigcup_{i\geq l} R_i(f_2)| &= \sum_{i=l}^{n^6+2n^5\log n} |R_i(f_2)| \\ &\leq \sum_{i=l}^{n^6+2n^5\log n} 2^{-i}|D_i(f_2)|. \end{aligned}$$

Now we must bound $|D_i(f_2)|$. For all y, for random f_2, $\log|S_{f_2}(y)|$ is just the random variable Y we have already considered. In particular, by Chernoff bounds, we can show

$$\Pr[|Y - E[Y]| > a] < e^{-a^2/4n^7}.$$

However, what we are interested in is closer to the case where the hash functions are fixed and y is chosen at random. To get the bound we desire, consider pairs $<y, f_2>$. Using the bound above, we can show, for any ϵ, for all but an ϵ fraction of the f_2's, the fraction of y's with $|S_{f_2}(y)| < 2^{E[Y]-a}$ is at most $e^{-a^2/4n^7}/\epsilon$. In particular, fix $\epsilon = 2^{-n/100}$. This gives that

$$\begin{aligned} &\sum_{i=l}^{n^6+2n^5\log n} 2^{-i}|D_i(f_2)| \\ &\leq \sum_{i=l}^{n^6+2n^5\log n} 2^{n^6+2n^5\log n-i+n/100} e^{-(E[Y]-i)^2/4n^7} \\ &\leq \sum_{j=E[Y]-n^6-2n^5\log n}^{E[Y]-l} 2^{n^6+2n^5\log n-E[Y]+j+n/100} e^{-j^2/4n^7} \end{aligned}$$

$$\begin{aligned}
&= 2^{n^6+2n^5\log n-E[Y]+n/100} \sum_{j=E[Y]-n^6-2n^5\log n}^{E[Y]-l} 2^{j-j^2/4n^7} \\
&\leq 2^{n^6+2n^5\log n-E[Y]+n/100+2} 2^{E[Y]-l-(E[Y]-l)^2/4n^7} \\
&\leq 2^{n^6+2n^5\log n-l+n/100+2} 2^{-(n^4/3)^2/4n^7} \\
&\leq 2^{n^6+2n^5\log n-l-n/60}.
\end{aligned}$$

The second case we consider is $z \in \bigcup_{i<l} R_i(f_2)$. Here, we will bound the number of such z in the range of f_3 by the number of x in the domain of f_3 which map to them. To do this, we first note that

$$\begin{aligned}
\bigcup_{i<l} |D_i(f_2)| &\leq \sum_{i=0}^{l} 2^{n^6+2n^5\log n+n/100} e^{-(E[Y]-i)^2/4n^7} \\
&\leq 2^{n^6+2n^5\log n+n/100+1} e^{-(E[Y]-l)^2/4n^7} \\
&\leq 2^{n^6+2n^5\log n+n/100} e^{-(n^4/3)^2/4n^7} \\
&\leq 2^{n^6+2n^5\log n-n/60}.
\end{aligned}$$

Now we can apply Lemma 2.2.10 to bound the number of x in the domain of f_3 which map to $z \in \bigcup_{i<l} R_i(f_2)$ by $2^{n^6+2n^5\log n-l-n/80-1}$ for almost all h_3.

Adding these two parts together, we get the desired bound on the range of f_3 for almost all hash functions, so h_4 is very unlikely to induce a collision. □

Lemma 4.3.12 *Suppose $l \leq E[Y] - n^4/3$. Then if there exists a non-trivial sibling finder A for $F_{4,l}$ then there exists a non-trivial sibling finder B for $F_{3,l}$.*

Proof First, B runs A to get x and outputs it. Next, given a random $f_3 \in F_{3,l}$, B picks a random h_4 and gives $f_4 = h_4 \circ f_3$ to A. Finally, B gets y from A and outputs it. If A gives a non-trivial sibling y of x under f_4, by Lemma 4.3.11, with overwhelming probability, it is a non-trivial sibling of x under f_3. □

Lemma 4.3.13 *If f is a one-way function and $E[Z] + n^4/4 \leq l \leq E[Y] - n^4/3$ then $F_{4,l}$ is a family of one-way hash functions mapping $\{0,1\}^{n^6+2n^5\log n-l}$ to $\{0,1\}^{n^6+2n^5\log n-l-n/100}$.*

4.3.6 Putting Things Together

In the previous section we showed that, given any one-way function, if we could choose l properly, we could construct a family of one-way hash functions obtaining a slight compression. In this section, we show how to obtain greater compression and also remove the need to choose l.

We begin with a way to convert a family of one-way hash functions obtaining even one bit of compression into a family of one-way hash functions obtaining an arbitrarily large compression ratio. Our approach differs from that used by Naor and Yung [55]. They directly construct of a sequence of small-compression families of one-way hash functions, each with different input and output lengths. On the other hand, we require only a single family of one-way hash functions in our construction. This is very important to us—otherwise we would not be able to remove the dependence on l.

First, we show that running a polynomial number of copies of a one-way hash function in parallel yields a one-way hash function.

Lemma 4.3.14 *Let F be a family of one-way hash functions mapping n_1 bits to n_2 bits ($n_1 > n_2$). Then for any $a, b \in n_1^{O(1)}$, $G = F^a \cdot \{I_b\}$, where I_b is the identity function on $\{0,1\}^b$, is a family of one-way hash functions mapping $an_1 + b$ bits to $an_2 + b$ bits.*

Proof Assume algorithm A is a non-trivial sibling finder for G succeeding with probability n^{-c}. Then we can construct a non-trivial sibling finder B for F as follows. Algorithm B will run algorithm A to obtain $x = x_1 \cdot \ldots \cdot x_a \cdot x_{a+1}$. It will then select an i at random between 1 and a, output x_i, and attempt to find a hard sibling for x_i. Given $f^i \in F$ picked at random, we pick f^j for $i \neq j$ at random from F and run A to get $y = y_1 \cdot \ldots \cdot y_a \cdot y_{a+1}$, and output y_i. If y is a non-trivial sibling of x under f_2, then for some j, $1 \leq j \leq a$, y_j must be a non-trivial sibling of x_j under f. Since i was chosen at random, there is an $1/a$ chance that $i = j$. Thus with probability at least $1/an^c$, we output a hard sibling for x_i. □

Next, we show taking the composition of a sequence of one-way hash functions yields a one-way hash function.

Lemma 4.3.15 *Let $n_1 \leq n_2 \leq \cdots \leq n_{m+1} = n_2^{O(1)}$. Suppose $\{G_i\}_{i=1}^m$ is a collection of families of one-way hash functions, where G_i maps n_{i+1} bits to n_i bits. Then $G = G_1 \circ G_2 \circ \cdots \circ G_m$ is*

a family of one-way hash functions mapping n_{m+1} bits to n_1 bits.

Proof Assume algorithm A is a non-trivial sibling finder for G succeeding with probability n^{-c}. Then we can construct a collection of probabilistic polynomial-time algorithms $\{B_i\}$ such that, for some i, B_i is a non-trivial sibling finder for G_i. Algorithm B_i will run algorithm A to obtain x. It will then select $g_j \in G_j$ at random for $j \neq i$, and let $g' = g_{i+1} \circ g_{i+2} \circ \cdots \circ g_m$ and $g'' = g_1 \circ g_2 \circ \cdots \circ g_{i-1}$. Next, it will compute and output $x' = g'(x)$. Getting a random $g_i \in G_i$, it will give $g'' \circ g_i \circ g'$ to A and get back y. Finally, it will output $g'(y)$.

If y is a non-trivial sibling of x under $g_1 \circ g_2 \circ \cdots \circ g_m$, then there is a unique j such that $(g_{j+1} \circ g_{j+2} \circ \cdots \circ g_m)(y)$ is a non-trivial sibling of $(g_{j+1} \circ g_{j+2} \circ \cdots \circ g_m)(x)$ under g_j. Thus if A finds a non-trivial sibling with probability n^{-c}, then some B_i must find a non-trivial sibling with probability $1/an^c$. □

The above two lemmas allow us virtually arbitrary compression given any one-way hash function.

Lemma 4.3.16 *Suppose F is a family of one-way hash functions mapping n_1 bits to n_2 bits. Then for any $n_3 > n_4 \geq n_1 - 1$, $n_3 = n_1^{O(1)}$, there exists a family of one-way hash functions mapping n_3 bits to n_4 bits.*

Proof Follows from Lemmas 4.3.14 and 4.3.15. □

Corollary 4.3.17 *Suppose $0 \leq l \leq 2n^6$. Then there exists a family of functions $F_{5,l}$ from $\{0,1\}^{96n^9}$ to $\{0,1\}^{2n^6}$ such that if f is a one-way function and $E[Z] + n^4/4 \leq l \leq E[Y] - n^4/3$ then $F_{5,l}$ is a family of one-way hash functions.*

Now we can finally construct a family of one-way functions based only on the assumption that f is one-way. Note that, since $E[Y] - E[Z] \geq 2n^4/3$, we only need l to be in a range of size at least $n^4/12$. Also, we know that l is between 0 and $2n^6$. Therefore, we know that one of $F_{5,n^4/12}, F_{5,2n^4/12}, \ldots, F_{5,2n^6}$ is a family of one-way hash functions. We construct F_6 as the set of functions f_6 mapping x to $f_{5,n^4/12}(x) \cdot f_{5,2n^4/12}(x) \cdot \ldots \cdot f_{5,2n^6}(x)$, where $f_{5,i} \in F_{5,i}$.

Lemma 4.3.18 *If f is a one-way function then F_6 is a family of one-way hash functions mapping $96n^9$ bits to $48n^8$ bits.*

Proof Assume probabilistic polynomial-time algorithm A is a non-trivial sibling finder for F_6. Then we will construct a non-trivial sibling finder B_l for $F_{5,l}$, where l is any multiple of $n^4/12$. In particular, we will be interested in such an l which is between $E[Z]+n^4/4$ and $E[Y]-n^4/3$, which is guaranteed to exist. Algorithm B_l simply gets x from A and outputs it. Then given a random $f_{5,l} \in F_{5,l}$, B_l randomly chooses the other $f_{5,i}$'s and gives the resultant f_6 to A. It then gets y from A. If y is a non-trivial sibling of x under f_6 then it is a non-trivial sibling of x under each $f_{5,i}$, and in particular under $f_{5,l}$. □

Summing up, we get the following theorems.

Theorem 4.3.19 *Under the assumption that one-way functions exist, one-way hash functions exist.*

Theorem 4.3.20 *Under the assumption that one-way functions exist, there exists a signature scheme which is secure against existential forgery under adaptive chosen message attacks.*

Finally, we note that, although this paper has been mostly phrased in terms of the uniform model of security, our construction works equally well in the non-uniform model. Thus we get the following theorem.

Theorem 4.3.21 *Under the assumption that one-way functions in the non-uniform model exist, there exists a signature scheme which is secure against existential forgery under adaptive chosen message attacks by polynomial-sized circuits.*

Bibliography

[1] M. Ajtai, J. Komlos, and E. Szemeredi. Deterministic simulation in logspace. In *Proceedings of the* 19^{th} *Annual ACM Symposium on Theory of Computing*, pages 132–140, May 1987.

[2] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:567–583, 1987.

[3] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Compact distributed data structures for adaptive routing. In *Proceedings of the* 21^{st} *Annual ACM Symposium on Theory of Computing*, pages 479–489, May 1989.

[4] B. Awerbuch, A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In *Proceedings of the* 16^{th} *Annual ACM Symposium on Theory of Computing*, pages 249–257, April 1984.

[5] L. Babai. Trading group theory for randomness. In *Proceedings of the* 17^{th} *Annual ACM Symposium on Theory of Computing*, pages 421–429, May 1985.

[6] L. Babai and S. Moran. Arthur Merlin games: A randomized proof system and a hierarchy of complexity classes. Manuscript.

[7] J. Beck and T. Fiala. Integral approximation sequences. *Mathematical Programming*, 30:88–98, 1984.

[8] J. Beck and J. Spencer. "Integer-making" theorems. *Discrete Applied Mathematics*, 3:1–8, 1981.

[9] M. Bellare. Private communication, 1989.

[10] M. Bellare and S. Goldwasser. Saving randomness in interactive proofs. Manuscript, November 1989.

[11] M. Bellare and S. Micali. How to sign given any trapdoor function. In *Proceedings of the* 20^{th} *Annual ACM Symposium on Theory of Computing*, pages 32–42, May 1988.

[12] M. Bellare and J. Rompel. Randomness-efficient sampling of arbitrary functions. Technical Memo MIT/LCS/TM–433, Lab. for Computer Science, MIT, Cambridge, MA, July 1990.

[13] B. Berger. Data structures for removing randomness. Technical Report MIT/LCS/TR–436, Lab. for Computer Science, MIT, Cambridge, MA, December 1988.

[14] B. Berger and J. Rompel. Simulating $(\log^c n)$-wise independence in NC. In *Proceedings of the* 30^{th} *Annual Symposium on Foundations of Computer Science*, pages 2–7. IEEE, October 1989. Also *Journal of the ACM*, to appear.

[15] B. Berger, J. Rompel, and P. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. In *Proceedings of the* 30^{th} *Annual Symposium on Foundations of Computer Science*, pages 54–59. IEEE, October 1989.

[16] B. Berger and P. Shor, September 1988. Unpublished note.

[17] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, November 1984.

[18] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. In *Proceedings of the* 18^{th} *Annual ACM Symposium on Theory of Computing*, pages 273–282, May 1986.

[19] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[20] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. In *Proceedings of the* 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 539–549. IEEE, October 1989.

[21] B. Chor and O. Goldreich. On the power of two–point based sampling. *Journal of Complexity*, 5:96–106, 1989.

[22] B. Chor, O. Goldreich, J. Hastad, J. Friedman, S. Rudich, and R. Smolensky. The bit extraction problem or t-resilient functions. In *Proceedings of the* 26^{th} *Annual Symposium on Foundations of Computer Science*, pages 396–407. IEEE, October 1985.

[23] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.

[24] A. Cohen and A. Wigderson. Dispersers, deterministic amplification, and weak random sources. In *Proceedings of the* 30^{th} *Annual Symposium on Foundations of Computer Science*, pages 14–19, October 1989.

[25] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM Journal on Computing*, 11:540–546, 1982.

[26] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[27] P. Erdos and D. Kleitman. On coloring graphs to maximize the proportion of multicolored k-edges. *Journal of Combinatorial Theory*, 5(2):164–169, September 1968.

[28] H. N. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM Journal on Computing*, 11:117–129, 1982.

[29] O. Goldreich. Private communication, June 1990.

[30] O. Goldreich, H. Krawczyk, and M. Luby. On the existence of pseudorandom generators. In *Proceedings of the* 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 12–24, October 1988.

[31] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proofs. *SIAM Journal on Computing*, 18(1):186–208, February 1989.

[32] S. Goldwasser, S. Micali, and R. Rivest. A secure digital signature scheme. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[33] S. Goldwasser and M. Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the* 18^{th} *Annual ACM Symposium on Theory of Computing*, pages 59–68, May 1986.

[34] J. Hastad. Pseudo-random generators under uniform assumptions. In *Proceedings of the* 22^{nd} *Annual ACM Symposium on Theory of Computing*, pages 395–404, May 1990.

[35] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, March 1963.

[36] R. Impagliazzo, L. Levin, and M. Luby. Pseudorandom generation from one-way functions. In *Proceedings of the* 21^{st} *Annual ACM Symposium on Theory of Computing*, pages 12–24, May 1989.

[37] R. Impagliazzo and D. Zuckerman. How to recycle random bits. In *Proceedings of the* 30^{th} *Annual Symposium on Foundations of Computer Science*, pages 248–253, October 1989.

[38] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[39] J. Kahn, G. Kalai, and N. Linial. The influence of variables on Boolean functions. In *Proceedings of the* 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 68–80. IEEE, October 1988.

[40] H. J. Karloff and J. Naor, October 1988. Private communication.

[41] H. J. Karloff and D. B. Shmoys. Efficient parallel algorithms for edge coloring problems. *Journal of Algorithms*, 8:39–52, 1987.

[42] R. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, New York, NY, 1975.

[43] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32(4):762–773, October 1985.

[44] L. Lamport. Constructing digital signatures from one-way functions. Technical Report CSL-98, SRI International, October 1979.

[45] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, 30:93–100, 1981.

[46] L. Levin. One-way functions and pseudorandom generators. In *Proceedings of the* 17^{th} *Annual ACM Symposium on Theory of Computing*, pages 363–365, May 1985.

[47] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[48] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, November 1986.

[49] M. Luby. Removing randomness in parallel computation without a processor penalty. In *Proceedings of the* 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 162–173. IEEE, October 1988.

[50] R. Merkle. A certified digital signature. Manuscript, 1979. Appears in CRYPTO '89 pp. 218–238.

[51] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptography (CRYPTO 87)*, pages 369–378. Springer–Verlag, August 1987.

[52] R. Merkle. One-way hash functions and DES. In *Advances in Cryptography (CRYPTO 89)*, pages 428–446. Springer–Verlag, August 1989.

[53] R. Motwani, J. Naor, and M. Naor. A generalized technique for derandomizing parallel algorithms, January 1989. Unpublished manuscript.

[54] R. Motwani, J. Naor, and M. Naor. The probabilistic method yields deterministic parallel algorithms. In *Proceedings of the* 30^{th} *Annual Symposium on Foundations of Computer Science*, pages 8–13. IEEE, October 1989.

[55] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the* 21^{st} *Annual ACM Symposium on Theory of Computing*, pages 33–43, May 1989.

[56] M. O. Rabin. Efficient dispersal of information for security load balancing and fault tolerance. *Journal of the ACM*, 1988. To appear.

[57] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37(4):130–143, October 1988.

[58] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, pages 120–126, February 1978.

[59] N. Sauer. On the density of families of sets. *Journal of Combinatorial Theory*, A:13:145–147, 1972.

[60] J. Spencer. Balancing games. *Journal of Combinatorial Theory*, B:23:68–74, 1977.

[61] J. Spencer. *Ten lectures on the probabilistic method.* SIAM, Philadelphia, PA, 1987.

[62] V. N. Vapnik and A. J. Chervonenkis. On uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[63] J. Vitter and J-H. Lin. Learning in parallel. In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 106–124, Cambridge, MA, 1988. Morgan Kaufmann.

[64] J. Vitter and J-H. Lin, May 1989. Private communication.

[65] V. G. Vizing. On the estimate of the chromatic class of a P-graph. *Diskret. Anal.*, 3:25–30, 1964. In Russian.

[66] A.C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the* 23^{rd} *Annual Symposium on Foundations of Computer Science*, pages 82–91, October 1982.